Java Fundamentals

HSBC Technology and Services

HSBC
The world's local bank

Agenda

- What is Inheritance?
- Inheritance Terminology
- Furniture Hierarchy
- java.lang.Object
- Polymorphism
  - Overloading
  - Overriding
    - toString
    - Constructor
- instanceof
- Constants

Agenda…

- Abstract
  - Methods
  - Classes
- Final
  - Classes
  - Methods
  - Data
- Abstract Vs final
- Interfaces
  - Abstract methods
  - final data
- Interfaces Vs Abstract classes

Inheritance

HSBC Technology and Services

HSBC
The world's local bank

Inheritance

- The Java programming language takes advantage of familiar, real life, ideas and concepts.
  - As we saw in the previous chapter, classes are the programming reflection of real world entities.
  - With the inheritance notion we continue this analogy.
- Just like a child inherits its parents' (and their ancestors') genetic characteristics, a class inherits its parent state and behavior.
  - State – data members.
  - Behavior – methods.

Inheritance Terminology

- Some terminology is needed to translate known concepts to Java programming concepts:
  - class – a data type.
  - superclass – the parent, also referred to as the "base" class.
  - extends – the inheritance relation
  - subclass – the child, the class that inherits (extends) the superclass.

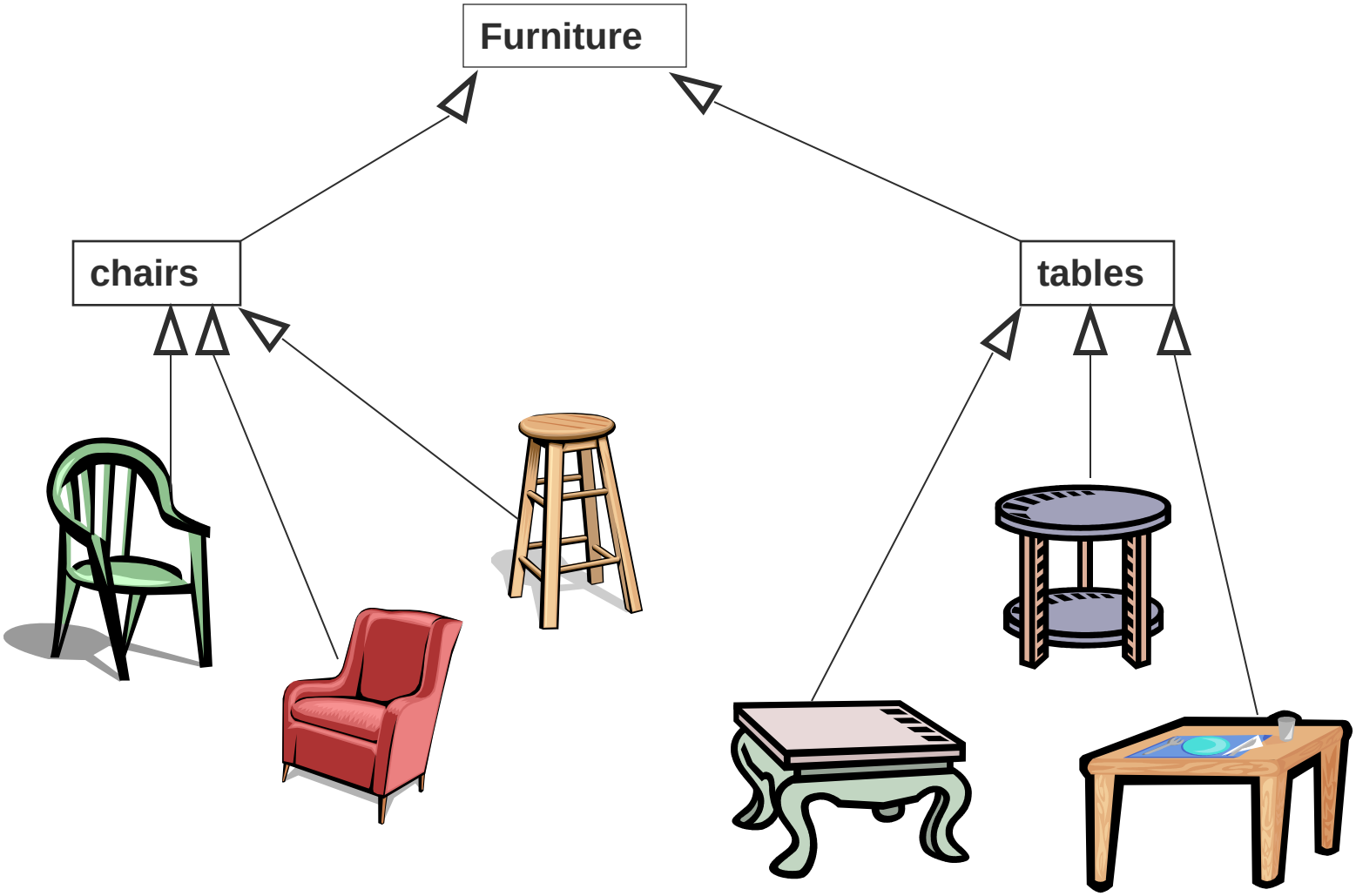
- Java Syntax:

subclass

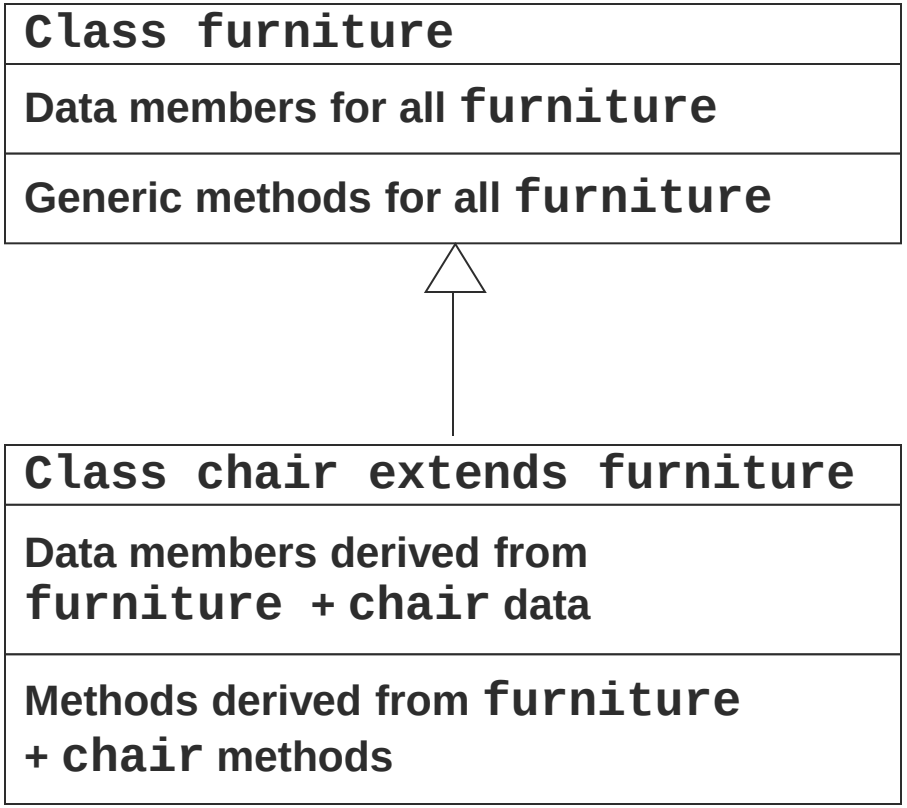
```
class chair extends furniture
{
}
```

superclass

The Furniture Dynasty



Furniture and Chair



Family Ties

```
class Furniture
int       m_nWidth
int       m_nHeight
String    m_sColor
int       m_nPrice
void setPrice(int price)
int  getPrice()
```

Unlike in real life, a class can (usually) choose its parent.

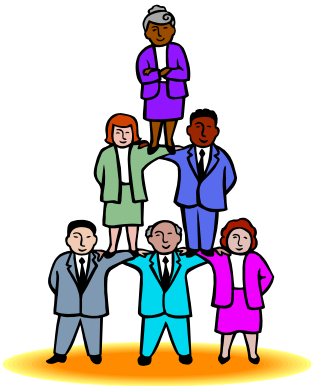
```
class Chair extends Furniture

int       m_nlegs
void adjustHeight(int num);
```

Class chair does not need to redefine `m_nWidth, m_nHeight` or any other data member or method belonging to its parent, furniture. Nevertheless, it is free to use them.

Inheritance Hierarchy

- Java is an Object-Oriented language so it uses inheritance to create a hierarchy model of the conceptual spectrum of the application.
  - As you go deeper inside the inheritance tree, specialization increases.
  - Sticking to the furniture example:
    - An adjustable student chair class will have a lot more features defining it that a mere furniture class.



Inheritance Advantages / Disadvantages

- Inheritance advantages:
  - Avoiding code cloning.
    - No need to re-implement furniture identifiers (Height, Width, etc.) in class chair.
  - Every change in the base class is being immediately reflected in all the classes derived from it.
- Inheritance disadvantages:
  - You may also inherit BUGS.
    - Before software can be reusable it first has to be usable- J

java.lang.Object

- No class is an orphan.
  - Every class has a parent class.
- When the term **`extends`** is omitted, the class is not parent-less, it inherits **`java.lang.Object`**.
- Class **`java.lang.Object`** is located at the root of the class hierarchy.
  - Every class has **`Object`** as a superclass either immediately (a parent) or mastering its ancestors hierarchy.
- Example class TextArea:

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.TextComponent
              |
              +--java.awt.TextArea
```

Everyone Inherits Object

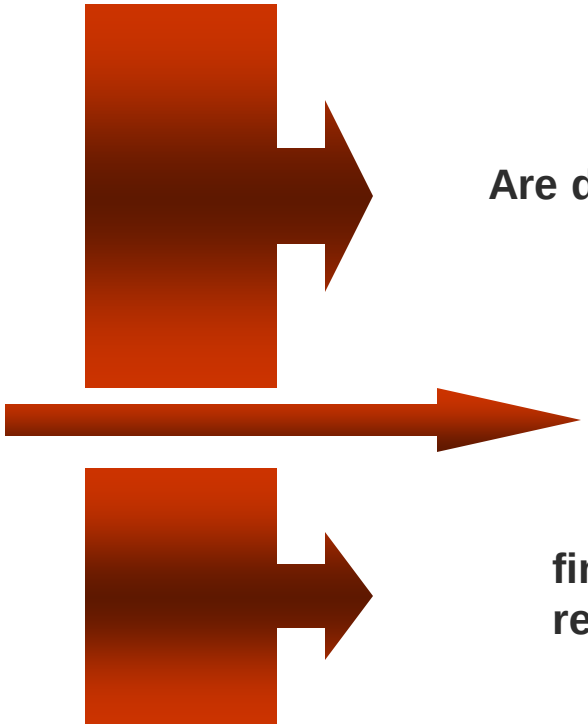
- The next example is a snippet of the output of the javap utility running the following code:

```
class Child
{
  public static void main(String args[])
   {
    System.out.println("I'm the direct child of Object");
   }
}
```

```
C:\Temp\JavaExamples\chapter4>javap -c Child
Compiled from child.java
class Child extends java.lang.Object {
    Child();
    public static void main(java.lang.String[]);
}
```

Class Object Methods

- All Java's objects implement class Object's methods:
  - `equals`
  - `hashCode`
  - `toString`
  - `clone`
  - `finalize`
  - `getClass`
  - `wait`
  - `notify`
  - `notifyAll`



Are designed to be overridden

Will be discussed in relation to reflection

final methods - will be discussed in relation to threads

Polymorphism

- Polymorphism is a complicated and impressive name meant to describe a very simple idea:
  - Using the same method name for different kinds of behavior.
- There are two types of polymorphism in Java:
  - Overloading – was preliminary discussed in the previous chapter.
  - Overriding

Overloading

- Overloading is about declaring some methods, in the same class, with the same name but with a different signature.
  - The signature is determined by the type, number and order of the arguments passed to the method.
- It is highly recommended that you will only overload methods performing related behavior.
- Class String contains a couple of examples for overloading:

```
public String substring(int beginIndex)

public String substring(int beginIndex, int endIndex)
```

Both method are called `subString`. they differ in their signature.

The compiler tests the parameters to decide which method is intended to be used.

Overloading in Class Furniture

```
class Furniture
{
 int m_nWidth;
 int m_nHeight;
 int m_nPrice;
 public void setPrice(int price)
 {
    m_nPrice = price;
 }
 public void setPrice(int price,int discount)
 {
    m_nPrice = price-(price * discount)/100;
 }
```

Overriding

- Overriding is a bit more complicated.
  - A sub class can own a method with the same signature (name and arguments) as its super class.
  - The method in the sub class is overriding, taking the place, of the one in the super class.
- The correct method is invoked at run time depending on the object's type.
  - The run time environment is performing dynamically binding (also called late or delayed binding) which allows it to act according to the object it handles.

Overriding `setPrice`

```
class Chair extends Furniture
{
  private int   m_nlegs;
  private boolean m_bUpholstery;
  private boolean m_bClubMember;
  public void setPrice(int price) {
   if (bClubMember)
     m_nPrice = price – 10;
   else
     m_nPrice = price
  }
  void adjustHeight(int num) {
    m_nHeight+=num;
  }
}
```

Overriding `Object`'s `toString()`

- Although **`java.lang.Object`** provides an implementation to the **`toString`** method, it is highly recommended to override it.
  - The original implementation consists of the class's name followed by the "at" sign (@) and the class's hexadecimal hash code.
    - Something like: "course@f032585a"
  - There is nothing wrong with this, except for it not being very user-friendly or informative.
    - Getting something like: "Java course for C++ programmers" is much more helpful for a class user.

Overriding `Object`'s `toString()`

- The return value of the overridden **`toString`** should be self-explanatory.
  - It should return all (or most) of the important information contained in the class.
- It is also recommended that you not force the class users to solely depend on that method for gathering information about your class.
  - Don't forget to provide your class with access methods for these important members.
- To invoke the **`toString()`** methods, you can explicitly call it or just pass your object into **`System.out.println`**.

toString Example

```
class Course
{
  String m_sName;
  String m_sDesc;
  public Course(String name,String desc)
  {
     m_sName = name;
     m_sDesc= desc;
  }
  public String getName()
  {
     return m_sName;
  }
  public String getDesc()
  {
     return m_sDesc;
  }
  public String toString()  {
    return m_sName +
         " course " + m_sDesc;
  }
}
```

```
class Main
{
  public static void main(String args[])
  {
    Course c = new Course("Java","c++
    programmers");
    System.out.println("registering to: "
    + c);
  }
}
```

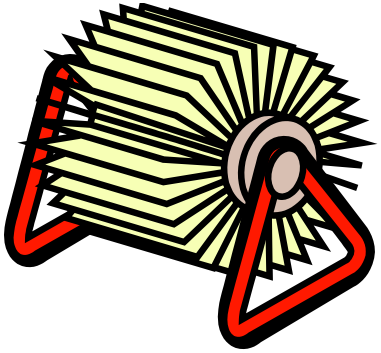
Overriding `equals`

- We have already used it with the `String` object.
- The equals method's role is to indicate whether some other object is "equal to" this one.
- The implementation supplied by **`java.lang.Object`** returns true if, and only if, the two compared referenced checked refer to the same object.
- It is recommended to override equals when the class user is interested to know whether the referenced classes are logically equivalent and whether or not they refer to the same object.

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

Overriding Exercise

- Create a class named Contact with the following members:
  - String m_sFirstName
  - String m_sLastName
  - String m_sPhone
  - Address m_address
    - String m_sStreet
    - String m_sHouseNum
    - String m_sCity
- Override **`equals`** and compare instances of class Contact.



Constructors and Inheritance

- When a sub class is instantiated, it always invokes the constructor of its super class.
  - It does so even before invoking its own constructor.
- Sometimes this default behavior is not enough, and you might wish to invoke your super class constructor explicitly.
  - This is essential when arguments are to be passed to it!
  - If this is the case, you will do so using the keyword **`super`**.

Invoking a Super-class Constructor

```
class Chair extends Furniture
{
  private int     m_nlegs;
  private boolean m_bUpholstery;
  public chair()
  {
     super("wood"); // Call the parent's Ctor
     System.out.println("Inside chair constructor");
  }
  void adjustHeight(int num)
  {
    m_nHeight+=num;
  }
}
```

Class `Chair` is now invoking a `Furniture` constructor that expects a string.

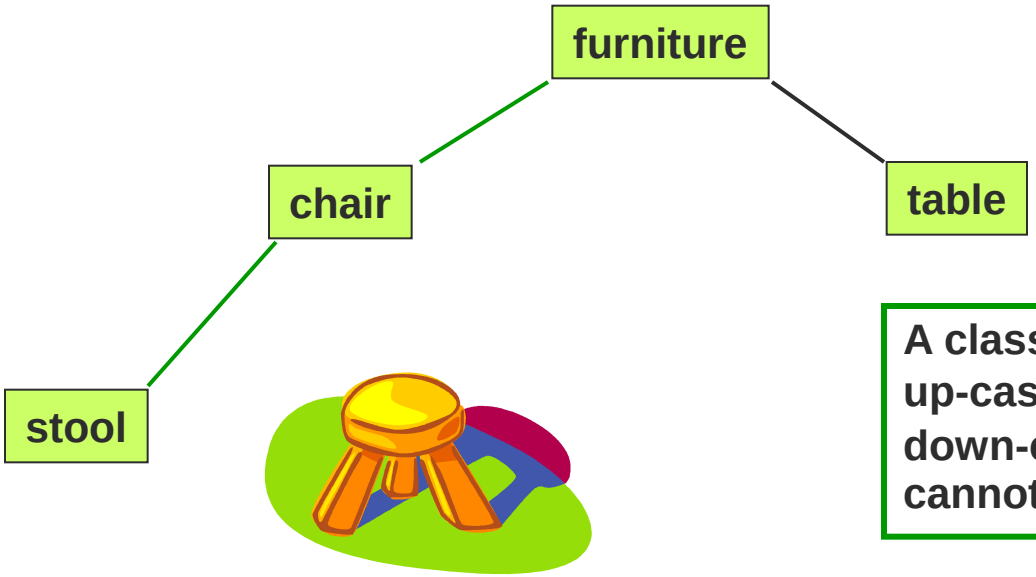
`super` should be the first line in the child class constructor.

Unveiling the Shadows

- The **`super`** keyword is also used to approach fields declared in the super class.
  - Class **`Chair`** can access the data members of class **`Furniture`** using the following syntax:
    - **`super.m_nHeight = 5;`**
  - The statement shown here is correct but yet verbose.
    - Class **`Furniture`** contains a data member by this name but class **`Chair`** does not, so it can be accessed without using **`super`**.
  - The use of **`super`** is unavoidable when a sub class is using an identically named data member as its parent or one of its ancestors.
    - Note that you cannot use **`super.super.super.member-name`** **……**

Casting

- Performing a casting action on an object means that you convert that object to another kind of object.
  - Casting can only be performed along the hierarchy tree, you cannot cast it "to the sides“.



A class of type `Chair` can be up-cast to `Furniture` and down-cast to stool, but it cannot be cast to a `Table` .....

Up - casting

- **Up-casting** is performed when a general object holds a more specialized one.
- It is completely legal to do the following casting:

```
Furniture MyFurniture = new Chair();
```

  - In this case we can only access, through **`MyFurniture`**, methods and data members declared in the super class, **`Furniture`**.
  - This kind of casting is very useful when you cannot be sure, in advance, exactly which one of **`furniture`** descents you will be instantiating.
- The other way around is, of course, illegal:

```
Chair MyChair = new Furniture();
```

Up – casting Example

```
class MyCast
{
  public static void main(String args[])
  {
     Furniture f = new Chair();
     f.setPrice(5);
     //f.adjustHeight(5);
  }
}
```

Up-casting: you can always simply do parent = child. (The other way around is a little more complicated.)

`setPrice()` is declared in class `Furniture` (the super class) so it can be accessed.

`adjustHeight(..)` cannot be accessed because it is not declared in the general class.

Down - casting

- **Down-casting** is a bit more complicated than up-casting:
  - It means to convert a general object to a more specialized one.
    - Note that both of them must be on the same hierarchy tree.
- In other words:
  - We expect that when we cast an object of type **`Furniture`** to **`Chair`**, it will have access to class **`Chair`** methods and data members.
  - This is only possible if is was initially referring to **`Chair`**:

```
Furniture F = new Chair();
Chair C = (Chair)F;
```

Down-cast Example

```
class MyCast
{
  public static void main(String args[])
  {
     Furniture F = new Chair();
     //F.adjustHeight(3);
     Chair C = (Chair)F;
     C.adjustHeight(3);
  }
}
```

Will not compile …

F can only access Furniture properties

After the down-cast, to Chair, the new reference can be used to freely access class Chair properties

No Sideways Casting

- As depicted a few slides ago, there are two classes that extend class **`Furniture`**: **`Chair`** and **`Table`**.
  - Casting can be made between **`Furniture`** and **`Chair`** but a **`Chair`** (as hard as it tries) will never become a **`Table`**.

```
Furniture F = new Table();

Chair C = (Chair)F;
```

Is legal and will compile.

Will compile but when checked, at run time, will throw a `ClassCastException`.

The Ultimate Ancestor

- Since **`java.lang.Object`** is the ancestor of all Java objects it can be assigned each one of them.
- Class **`Vector`** is a container for objects.
  - It will be discussed in detail in later chapters.
  - Here we will use 2 of its methods to demonstrate casting.

```
public void addElement(Object obj)
```

Expects an `Object` so we can practically supply it with any type of object

```
public Object elementAt(int index)
```

The method returns an object of type `Object` that must be explicitly down-cast to be used

MyVector.java

```
import java.util.Vector;
 class MyVector
{
  Vector m_vChairVect;
  public static void main(String args[])
  {
     MyVector chairVec = new MyVector();
  }
public MyVector()
  {
    m_vChairVect = new Vector(); // Create a new collection
    buildVec();
    changeHeight();
  }
```

Up and Down Casting

```
public void buildVec()
  {
    for (int i=0;i<5;i++){
      Chair c = new Chair();
      m_vChairVect.addElement(c);  // Add a new element
    }
  }
public void changeHeight() {
    for (int i=0;i<m_vChairVect.size();i++)
    {
       // Get an element out of the collection:
       Chair c = (Chair)m_vChairVect.elementAt(i);
       c.adjustHeight(i);
       System.out.println("the chair height is: " +
                                c.getHeight());
    }
  }
}
```

Safe Casting

- When performing a down-cast, it is essential to know how the object was declared in the beginning.
- In the last slide, we created objects of type **`Chair`** and placed them inside a **`Vector`**.

```
Chair c = new Chair();
m_vChairVect.addElement(c);
```

  - When we took them out, we received objects of type **`Object`** and performed a down-cast.

```
Chair c = (Chair)m_vChairVect.elementAt(i);
```

  - Neglecting to perform the correct cast (anything but **`Chair`**) would have been ended in an exception being thrown.
    - The obvious conclusion is that we must perform safe casting.

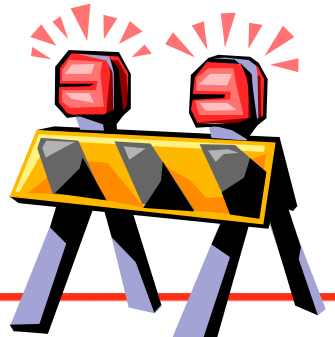
The `instanceof` Operator

- The `instanceof` operator is designed to compare a reference with a type.
  - So, in order to perform a safe cast, we have recoded the method **`changeHeight`**:

```
public void changeHeight()
  {
    for (int i=0;i<m_vChairVect.size();i++){
       Object obj = m_vChairVect.elementAt(i);
       if (obj instanceof Chair){  // Safe downcasting
         Chair c = (Chair)obj ;
          c.adjustHeight(i);
          System.out.println("the chair height is: " + c.getHeight());
         }
    }
  }
```

Blocking Inheritance

- A little earlier, we said that a class is lucky to pick its dream parent.
  - Well, it is time to add a little exception to this notion!
- When a class is declared **`final`** it cannot be extended.
  - If you try to extend it, you will end up with a run time error.
- What's the purpose of blocking inheritance?
  - When a class is being extended, the sub class does not only inherit data members and methods. It can also change them (a lot more about this will shortly follow).
    - There might be times when a class developer is not so liberal as to allow that.
    - Either the developer is restricted to some pattern or behavior that should not be changed – kind of a constant, really.



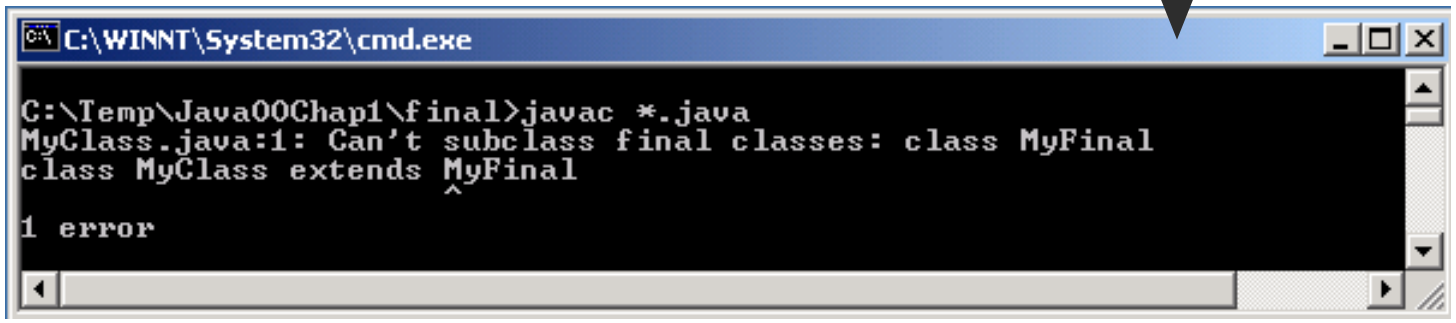
Final Classes

```
final class MyFinal
{
   public MyFinal()
  {
   System.out.println("I'm the ultimate class");
  }
}
```

This class is `final`

```
class MyClass extends MyFinal
{
}
```

This class tries to extend a `final` class. The result:



`final` Methods

- So, in order to prevent a class from being extended, we use the keyword **`final`**.
  - It is possible to apply a very similar behavior to a class's method.
  - When a method is declared **`final`** it cannot be overridden.
- Naturally, another type of methods that cannot be overridden are **`static`** methods.
  - Quite reasonable when you think about it:
    - A **`static`** method belongs to a class, not an instance, so there is never any ambiguity about its owner.

Abstract Class

- We have previously seen how a class author can block its creation from being extended.
- The keyword `abstract` is used to do the opposite:
  - An abstract class must be extended in order to create an instance.
  - An abstract class cannot be instantiated.
- You can think about an abstract class as a concept.
  - This does not prevent it from containing methods and data members.

Food and Omelet

```
abstract class Food
{
 int     m_nCalories;
 public int CaloriesPerAmount(int amount)
 {
  return (m_nCalories*amount)/100;
 }
 public void setCaloriesPer100gram(int cal)
 {
  m_nCalories = cal;
 }
}
```

```
class Omelet extends Food
{
}
```

Only `Omelet` can be instantiated, invoke class Food methods and use its data members.

Abstract Method

- An abstract method is an empty one, it does not include implementation.
  - A class containing at least one abstract method must be declared abstract.
  - However, an abstract class doesn’t have to make all its methods abstract !
- The extending class must handle it in one of two ways:
  - Implement the method.
  - Declare it the same way its super class has done, which will make it an abstract class too.
- A class extending an abstract class must be careful to implement all its super's abstract methods or else it would itself become abstract.

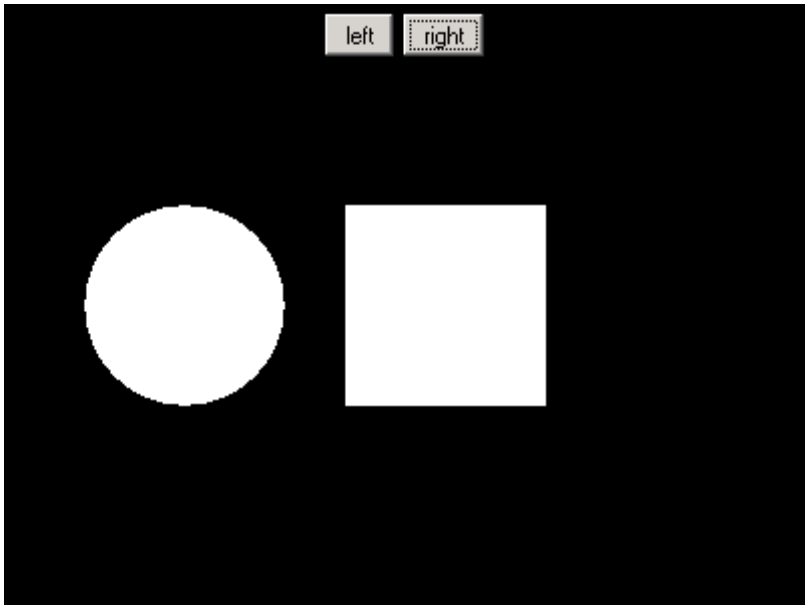
Abstract Method Example

```
abstract class Food
{
 abstract public void importantInfo();
}
```

```
class Omelet extends Food
{
  public void importantInfo()
  {
      System.out.println("Just throw some eggs,
       oil and salt into a frying pan");
  }
}
```

Abstract Class Example

- The following example contains one abstract class: **`Shape`** and two classes that inherit it: **`Cycle and Rectangle.`**
- Class Shape has two methods:
  - An abstract method **`draw`**.
    - Which is implemented in the extending classes.
  - A real method to **`move`** which is responsible for moving a shape left and right.



Class Board

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Board extends Applet
{
    Shape sArr[]=new Shape[2];
    Board brd;
    public void init()
    {
        brd=this;
        Implementor imp=new Implementor();
        this.setBackground(Color.black);
        this.setForeground(Color.white);
        Button bLeft=new Button(" left ");
        bLeft.addActionListener(imp);
        Button bRight=new Button(" right ");
        bRight.addActionListener(imp);
        this.add(bLeft);
        this.add(bRight);
```

The Action Listener

```
20.                 //creating the shapes that will be draw on the applet
21.                 sArr[0]=new Circle(120,100,100,100);
22.                 sArr[1]=new Rectangle(250,100,100,100);
23.         }
24.         public void paint(Graphics g){
25.                 for(int i=0;i<sArr.length;i++)
26.                 {
27.                         sArr[i].draw(g);
28.                 }
29.         }
30.         private class Implementor implements ActionListener  {
31.                 public void actionPerformed(ActionEvent e)
32.                 {
33.                         if(e.getActionCommand().equals(" left "))
34.                         {
35.                                 for(int i=0;i<sArr.length;i++)
36.                                 {
37.                                         sArr[i].move("left",brd);
38.                                 }
```

Class Board - cont'd

```
                        }
                       else
                       {
                               for(int i=0;i<sArr.length;i++)
                               {
                                   sArr[i].move("right",brd);
                               }
                       }
               }
      }
}
```

Class Shape

```
import java.awt.*;
abstract public class Shape
{
        int x,y,width,height;

        abstract public void draw(Graphics g);

        public void move(String direction,Board brd){
                if(direction.equals("right")&& x<=390)
                {
                        x+=10;
                }
                else{
                        if(direction.equals("left")&& x>=10)
                                x-=10;
                }
                brd.repaint();
        }
}
```

Class Circle

```
import java.awt.*;
public class Circle extends Shape
{
        public Circle(int x,int y,int width,int height)
        {
                this.x=x;
                this.y=y;
                this.width=width;
                this.height=height;
        }
        public void draw(Graphics g)
        {
                g.fillOval(x,y,width,height);
        }
}
```

Class Rectangle

```
import java.awt.*;
public class Rectangle extends Shape
{
        public Rectangle(int x,int y,int width,int height)
        {
                this.x=x;
                this.y=y;
                this.width=width;
                this.height=height;
        }
        public void draw(Graphics g)
        {
                g.fillRect(x,y,width,height);
        }
}
```

Summary

- Just like a child inherits its parents' (and their ancestors') genetic characteristics, a class inherits its parent state and behavior
- Inheritance Terminology
  - Super class
  - Sub Class
- java.lang.Object is ancestor for all
- Overloading
  - add(int a, int b)
  - add(String a, String b)
- Overriding
    - toString
- Calling super class Constructor using super
- final static String COLOR="RED"

Summary…

- Abstract
  - Methods
    - Only declaration
  - Classes
    - Can’t be instantiated
- Final
  - Classes
    - Cant be inherited
  - Methods
    - Cant be overridden
  - Data
    - Can’t be changed
- Abstract Vs final

A Single Parent

- Java does not support multiple inheritance.
  - Every sub class may have exactly one direct super class.
- The language designers have decided to leave out multiple inheritance because of the difficulties and complications it creates.
- Java has an elegant bypass to this matter, called **interface**.
  - By using interfaces you can gain some of the benefits of a super class, without having to deal with complications.

~~**class Chair extends Furniture , office-stuff**~~

Interfaces

- An **interface** is a skeleton of a class, it specifies a set of unimplemented methods and class variables.
  - The classes making use of the interface are responsible for implementing these methods.

The keyword `implements` is used to specify a connection between a class and an interface.

```
Class MyClass implements IMyInterface
{
   public void work() {
    System.out.println("I'm working very hard");
   }
 }
```

```
interface IMyInterface {
   public void work();
}
```

Multiple Interfaces

- The fact that the interface's methods are abstract (lack implementation) creates an elegant round-about to the multiple inheritance problem.
  - A class may extend only one class but may implement as many interfaces as it likes without worrying about names collision.

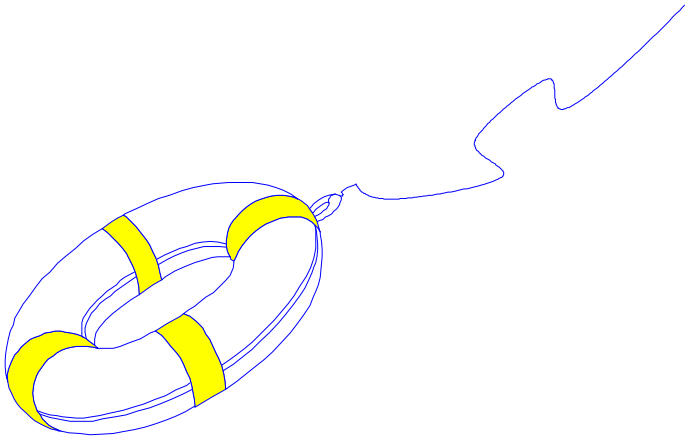
```
class MyClass extends YourClass implements IMyInterface,
IHerInterface, IHisInterface
 {
   . . . . . .
 }
```

Reusing Interfaces

- An interface may extend other interfaces.

```
interface IMyInterface extends Runnable, Serializable
```

Both Runnable and Serializable belong to the Java API



Named Constants

- The methods and the data members of an interface are actually constants, they are always **`public`**, **`static`** and **`final`**.
  - You don't need to include any of those keywords in the interface's code to make them like that.
- You can use an interface to declare a set of named constants:

```
class MyClass implements Isalaries {
   public static void main(String args[])
   {
      System.out.println("My salary is: "+ 2*minSalary);
   }
}
```

```
interface ISalaries {
    int minSalary = 500;
    int maxSalary = 50000;
}
```

Implementing Multiple Interfaces

```
class Person implements ISleep, IWork
{
  public Person()
  {
    TaskList task = new TaskList();
    task.add(this);
  }
  public void sleep()
  {
    System.out.println("I'm sleeping 12 hours a day");
  }
  public void work()
  {
    System.out.println("I'm working 4 hours a day");
  }
}
```

```
interface ISleep
{
  public void sleep();
}
```

```
interface IWork
{
  public void work();
}
```

The whole `Person` object is being sent to the `TaskList` object.

Main and TaskList Classes

```
class Main
{
  public static void main(String args[])
  {
    Person person = new Person();
  }
}
```

```
class TaskList
{
   IWork m_work;
   public void add(IWork work)
   {
     m_work = work;
     m_work.work();
   }
}
```

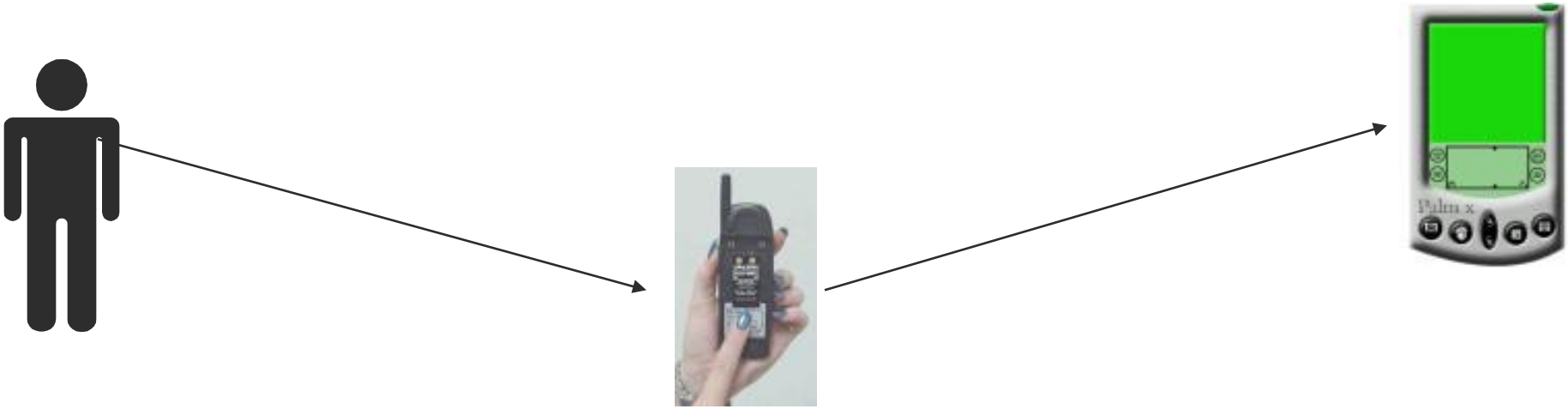
Inside `TaskList` we recognize only the part of `Person` which implements `Iwork`.

Interfaces vs. Abstract Classes

- Interfaces and abstract classes have a lot in common:
  - Specify a set of methods for an object.
  - Cannot be instantiated.

- However, don’t forget they also have 2 important differences:
  - While an interface can’t implement any of its declared methods, an abstract class may offer implementation.
  - Naturally, and abstract class prevents you from extending another class in the future.

Interfaces – Another Perspective



An interface is a named collection of method definitions (without implementations).

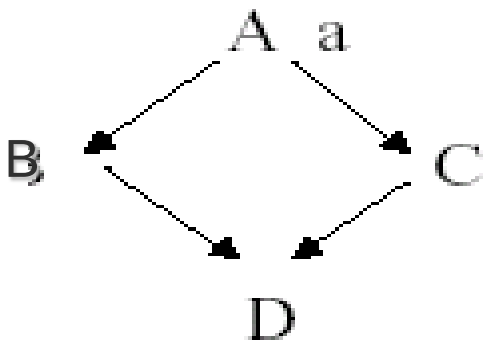
An interface defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.

An interface defines a set of methods but does not implement them.

Interfaces…

A technique to obtain multiple inheritance without problems of inheriting the same instance variable twice

- Contain only specifications, not code
- Can be implemented by many classes
- Provides for shared functionality
- All methods are public
- Also can be used to inherit constants



Example

```
class Circle extends Shape
        {       Point center;
                Integer radius;
        }
        public interface Scalable
        {       public void changeScale(int percent);}

class ScalableCircle extends Circle implements Scalable
                {
                                // changeScale() method must be implemented here
                                public void changeScale(int percent){. . .};
                }
```

(similarities)

A variable can be declared using *either* its **class** type *or* its **interface** type:

Scalable c = new ScalableCircle();

- Both can be extended:

```
public interface Readable
{
public String read();
}
public interface Searchable extends Readable
{
public int findPosition();
}
```

Differences

Interfaces cannot specify static methods

Interfaces cannot have instance variables

Interfaces cannot be instantiated using new

One class may implement multiple interfaces

```
public class Word implements Searchable, Replaceable
{
public int findPosition(Searchable s) {...}
public void replace(Replaceable w1,
Replaceable w2) {...}
}
```

}

Packages

Packages are

- Mechanism for partitioning the class name space into more manageable chunks
- Containers for classes that are used to keep the class name space compartmentalized
- Naming and visibility control mechanism

Defining Package

- Include *package* Command
- *It should be the first statement of java source file*.

File System Directories are used to store a package

Hence….

Folder should be created to store .class files with same name as of package Name

```
package myPackage;

public class ProtectionDemo
{
        ……
} // end class

subclass inside myPackage:

package myPackage;
public class subDemo extends ProtectionDemo
{
        public class User
                {
                        public void printStuff() {
                        ProtectionDemo d = new ProtectionDemo();
                        ….
                } end printStuff
} // end
```

}

Executing the package

Java mypack.user

Importing Packages

- To bring the certain classes or entire packages into visibility.

```
Import package.classname
```

}

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

HSBC

The world's local bank

About HSBC Technology and Services

HSBC Technology and Services (HTS) is a pivotal part of the Group and seamlessly integrates technology platforms and operations with an aim to re-define customer experience and drive down unit cost of production. Its solutions connect people, devices and networks across the globe and combine domain expertise, process skills and technology to deliver unparalleled business value, thereby enabling HSBC to stay ahead of competition by addressing market changes quickly and developing profitable customer relationships.

Presenter's Contact Details:
Name: Deepak Ratnani
Role: Team Leader
Direct: + 91-20 -66423608
Email: DeepakRatnani@hsbc.co.in