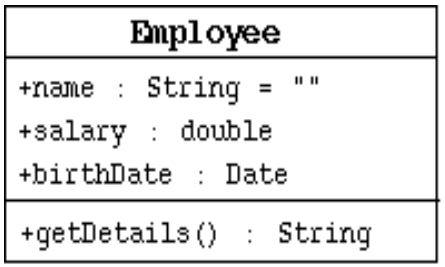
hi-tech college + JOHN BRYCE Leading in IT Education a matrix company

Class Design

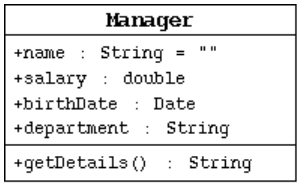
Inheritance and Polymorphism

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

The "*is a*" Relationship

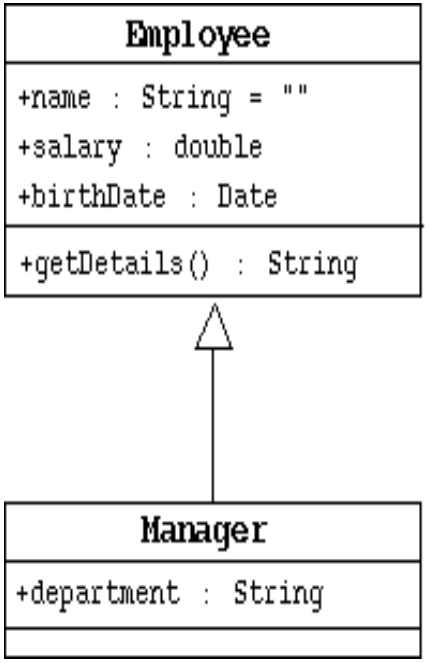



```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```




```
public class Manager {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String department;

  public String getDetails() {...}
}
```

The "*is a*" Relationship

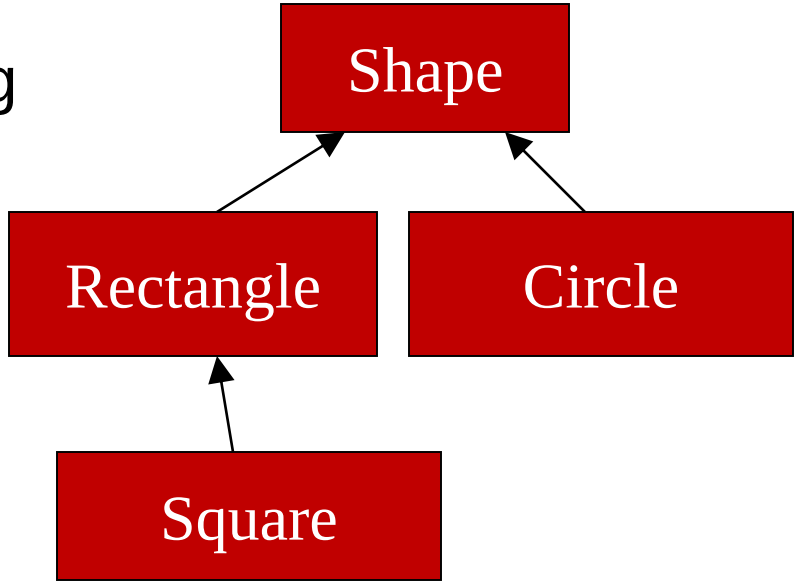



```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```

```
public class Manager extends Employee {
  public String department;
}
```

Class inheritance

- Java implements **single inheritance** (unlike C++ that has multiple inheritance) among classes
- Single inheritance makes code more reliable.
- *Interfaces* provide the benefits of multiple inheritance without drawbacks.



Inheritance example

```
public class Shape {
    protected double area;
    public double getArea() { return area; }
}
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
        area = Math.PI * radius * radius;
    }
}
```

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

Inheritance

- Keyword `extends` is used
- If the `extend` superclass clause is omitted, the class implicitly `extends` the class `java.lang.Object`
- Thus, `java.lang.Object` is the root of the class hierarchy, since every class is its subclass either directly or indirectly.

Inheritance

- A subclass inherits all members of its superclass, except those who are invisible to the subclass (`private`).
- Attributes of the subclass can hide members of the superclass. In this case the **`super`** pseudo variable is used to access those members.

Access Modifiers Summary

Modifier	Same Class	Same Package	Sub-Class	Universe
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	
default	Yes	Yes		
private	Yes			

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

Constructors

- When a subclass is instantiated the first thing that happens is that the superclass's constructors are called.
- If it is not specified explicitly, a parameterless constructor is called (if exists - otherwise compile time error arises)
- `this` can be used to explicitly specify which constructor of the same class is to be called
- `super` can be used to explicitly specify which constructor of the superclass is to be called

Example - this

```
public class Rectangle extends Shape {

    private double a,b;
    public Rectangle(double a, double b) {
        this.a = a; this.b = b;
        area = a*b;
    }
    public Rectangle(double a) {
        this(a,a);
    }
}
```

Example - super

```
public class Rectangle extends Shape {
    private double a,b;
    public Rectangle(double a, double b) {
        this.a = a; this.b = b;
        area = a*b;
    } // implicitly calling Shape()constructor
}
public class Square extends Rectangle {
    public Square(double a) {
        super(a, a);
    } // explicitly calling Rectangle(a,a)
  constructor.
}
```

Method overloading

- Methods with the same name may be written in one class or its subclasses as long as:
  - Arguments types are different

    or
  - Number of arguments is different

Example

```
public class Printer {

  public void print(int x){
          System.out.println(x);
   }
   public void print(int x,int y){
          System.out.println(x+y);
   }
   public void print(String str){
          System.out.println(str);
   }
   public String print(String str1, int x){
          System.out.println(str1+x);
         return str1+x;
   }
}
```

Method overriding

- Inherited methods can be overriden: subclass re-implements a method's body.
- Dynamic binding is used to call such methods.
- Class (static) members (neither variables nor methods) are NEVER inherited.
- Class methods are statically bound.
- Instance attributes are also hidden during inheritance - this can cause non-trivial errors

Example

```
public class Shape {
    public void paint() { // do nothing }
}
public class Circle extends Shape {
    public void paint() { // overriding
        … draw a circle
    }
}

…
Shape s = new Circle(5.0); // assume such constr.
s.paint() // a circle is painted
…
```

Method overriding

- The signature (argument list + (optionally) return type) of the overriding method MUST be identical to that of the overridden one.
- The subclass can declare the method with the same or less restrictive accessibility.
  This way the instances of the subclass can be safely used in place of the superclass's ones.
- For the same reason at most the exceptions declared in the superclass can be declared

super Example

```
public class Employee {
    …
    public String getDetails() {
        return name+” “+salary;
    }
}

public class Manager extends Employee{
    …
    public String getDetails() {
        return super.getDetails()
                +” “+dept;
    }
}
```

Rules About Overridden Methods

```
public class Parent {
  public void doSomething() {}
}
public class Child extends Parent {
  private void doSomething() {}
}
public class UseBoth {
  public void doOtherThing() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.doSomething();
    p2.doSomething();
}
```

Final classes and methods

- Class which declared to be final can not be extended.

e.g. `java.lang.String`

- Final methods can not be overridden

References types and inheritance

- A class-reference type variable can be assigned a reference to an instance of the declared class including to those that are instances of any sublclasses.

```
Shape s1 = new Shape();
Shape s2 = new Circle(5.0);
```

- As a result, `Object` type references can be assigned any instance references.

Polymorphism

- ***Polymorphism*** is the ability to have many different forms; for example, the Circle class has access to methods from Shape class.
- An *object* has only one form.
- A *reference* variable can refer to objects of different forms.

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

Polymorphism in OOP

- Method polymorhism: An overridden method has many implementations. It is determined dynamically which is used.
- Object polymorhism: A subclass has all the functionality of its superclass. Thus, an instance of a subclass can be used as same as where an instance of the superclass can be used.

Casting

- Use ***instanceof*** to test the type of an object
- Restore full functionality of an object by casting
- Check for proper casting using the following guidelines:
  - Casts up hierarchy are done implicitly
  - Downward casts must be to a subclass and is checked by the compiler
  - The object type is checked at runtime, while runtime errors can occur

Downcasting

```
Shape s;
Circle c = new Circle(1);
Rectangle r = new Rectangle(1.0, 2.0);
s = c; // polymorphism
s = (Shape) c; // needless casting

if(s instanceof Circle){
  Circle c1 = (Circle) s; // downcasting
  Circle c2 = (Circle) r; // illegal -
                          //impossible
}
```

Heterogeneous Collections

- Collections of objects with the same class type are called ***homogenous*** collections.
- MyDate[] dates = new MyDate[2];
- dates[0] = new MyDate(22, 12, 1964);
- dates[1] = new MyDate(22, 7, 1964);

- Collections of objects with different class types are called ***heterogeneous*** collections.
- Employee [] staff = new Employee[1024];
- staff[0] = new Manager();
- staff[1] = new Employee();
- staff[2] = new Engineer();

Object Methods Frequently Being Overridden

- Recall that the Object class is the root of all classes in Java
- A class declaration with no extends clause, implicitly uses "extends Object"
- Object's methods that should be overridden
  - equals
  - toString

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

hi-tech college
JOHN BRYCE
Leading in IT Education
a matrix company

The equals Method

- The == operator determines if two references are identical to each other (that is, refer to the same object)
- The *equals* method determines if objects are "equal" by their contents, but not necessarily identical (have the same reference)
- The Object implementation of the *equals* method uses the == operator
- User classes can override the *equals* method to implement a domain-specific test for equality

Equal objects in Java

- In case of reference type variables the operator `==` means that the two references are the same. `a!=b` is the same as `!(a==b)`
- Content based equality is implemented by overriding the `equals` method declared in `java.lang.Object`
- Thus `"a" == "a"` may be false, but `"a".equals("a")` is true.

The *equals* Method

```
public class MyDate {
  private int day;
  private int month;
  private int year;

  public MyDate(int day, int month, int year) {
    this.day   = day;
    this.month = month;
    this.year  = year;
  }

  public boolean equals(Object o) {
    boolean result = false;
    if ( (o != null) && (o instanceof MyDate) ) {
      MyDate d = (MyDate) o;
      if ( (day == d.day) && (month == d.month)
           && (year == d.year) ) {
        result = true;
      }
    }
    return result;
  }

  public int hashCode() {
    return (    (new Integer(day).hashCode())
              ^ (new Integer(month).hashCode())
              ^ (new Integer(year).hashCode())
           );
  }
}
```

```
public class TestEquals {
  public static void main(String[] args) {
    MyDate  date1 = new MyDate(14, 3, 1976);
    MyDate  date2 = new MyDate(14, 3, 1976);

    if ( date1 == date2 ) {
      System.out.println("date1 is identical to date2");
    } else {
      System.out.println("date1 is not identical to date2");
    }

    if ( date1.equals(date2) ) {
      System.out.println("date1 is equal to date2");
    } else {
      System.out.println("date1 is not equal to date2");
    }

    System.out.println("set date2 = date1;");
    date2 = date1;

    if ( date1 == date2 ) {
      System.out.println("date1 is identical to date2");
    } else {
      System.out.println("date1 is not identical to date2");
    }
  }
}
```

The *toString* Method

- Converts an object to a String
- Used during string concatenation
- Override this method to provide information about a user-defined object in a readable format
- Primitive types are converted to a String using the **wrapper** class's toString static method

Wrapper Classes

- Look at primitive data elements as objects
- Especially useful in heterogeneous collections
- Java `ArrayList` contains only Object s, so the only way to store *int*s in it, is to wrap them - treat them like objects

Primitive Data Type	Wrapper Class
boolean	Boolean
int	Integer
byte	Byte
char	Character
short	Short
long	Long
float	Float
double	Double

Wrapper Classes

- ArrayList list = new ArrayList();
- list.add (new Integer(5));
- list.add (new Integer(7));
- int k = ((Integer)list.get(0)).intValue();

Wrapper Class

- Also defines primitives related services such as:
  - parseInt , parseFloat, …
  - toString
  - Equals
  - Min & max values

References

- http://java.sun.com/javase/6/docs/technotes/guides
- SUN Educational Services SL-275
  download:
  http://www.sun.com/products-n-solutions/edu/programs/sai/download/SL275.E.2.desc.pdf