Lesson - 5: Inheritance & Interfaces- Day -2 & 3

Two Ways Inheritance Arises

- We saw `Manager` was a natural choice for a *subclass* of `Employee` because it *extends* `Employee's` behavior.

- Another situation that gives rise to inheritance occurs when several classes are seen to naturally belong to the same general type – this is *generalization.*

  In the "figures" example (from next slide), it seems natural to generalize the curves `Triangle`, `Circle`, `Square.`

- Demo `Package` : closedcurve.bad

```
final class Triangle {
        final double base;
        final double height;
Triangle(double base, double height){
                this.base = base;
               this.height=height;

        }
double computeArea() {
                return (0.5 * base * height);
        }
}

final class Square {
        final double side;
        Square(double side) {
                this.side = side;
        }
        double computeArea() {
                return(side*side);
        }
}

final class Circle {
        final double radius;
        Circle(double radius) {
                this.radius = radius;
        }
        double computeArea() {
                return (Math.PI * radius * radius);
                        }
                }
```

```
//Illustrates a non-OO (= bad) way of computing areas
class Test {
        public static void main(String[] args) {

Object[] objects = {new Triangle(5,5,5),

                new Square(3),

                new Circle(3)};
//compute areas
for(Object o : objects) {
 if(o instanceof Triangle) {

        Triangle t = (Triangle)o;

        System.out.println(t.computeArea());
                                                }
if(o instanceof Square) {

        Square s = (Square)o;
System.out.println(s.computeArea());
                                                }
if(o instanceof Circle) {

        Circle c = (Circle)o;
System.out.println(c.computeArea());
                                                }
                                        }
    }
        }
```

Points to observe

- Notice we can arrange Triangle, Square, Circle into an array of type Object[] by polymorphism. But Object does not have a computeArea() method, so we cannot polymorphically compute areas by using a single superclass method computeArea.
- Instead, if we use an array Object[], we have to repeatedly test the type of the Object in the area in order to execute the correct computeArea() method (using the instanceof operator).
- This approach needs improvement!
- Improvement shown by using Abstract class

Abstract Classes

- An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in subclasses.
- Some times you may create a class just to represent a concept rather than to represent the object.
- If a class is declared *abstract*, it cannot be instantiated.
- If a method is declared abstract, it cannot have a body -- it can only be declared.
- If a class has at least one abstract method, the class must be declared abstract.

Abstract Classes

- Abstract classes may include instance variables and other non-abstract (implemented) methods
- We use the modifier **`abstract`** on the class header to declare a class as abstract
- An abstract method is created by specifying the **abstract** type modifier. An abstract method contains no body and is, therefore, not implemented by the superclass.

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent
- An abstract method cannot be defined as final (because it must be overridden) or static (because it has no definition yet)
- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate

- Can **generalize** the behavior of these geometric shape classes to support *polymorphic* access to a general computeArea() method.

```
abstract class ClosedCurve {
        abstract double computeArea();
}
```

See Demo pack : closedcurve.good

Demo Code : AbstractDemo.java

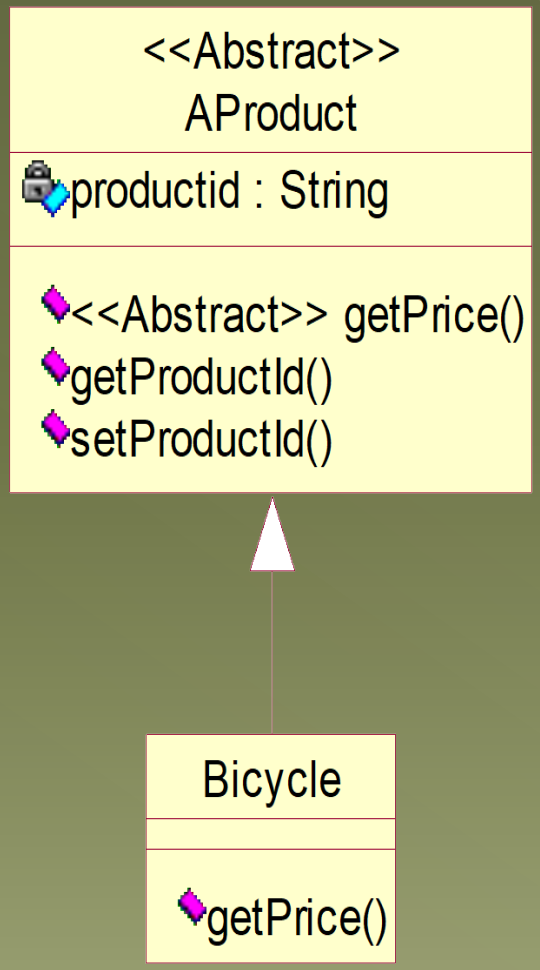
```
final class Triangle  extends ClosedCurve {
        final double base;
        final double height;
Triangle(double base, double height){
                this.base = base;
               this.height=height;


        }
double computeArea() {
                return (0.5 * base * height);
        }
}

final class Square extends ClosedCurve{
        final double side;
        Square(double side) {
                this.side = side;
        }
        double computeArea() {
                return(side*side);
        }
}

final class Circle extends ClosedCurve{
        final double radius;
        Circle(double radius) {
                this.radius = radius;
        }
        double computeArea() {
                return (Math.PI * radius * radius);
                        }
                }
```

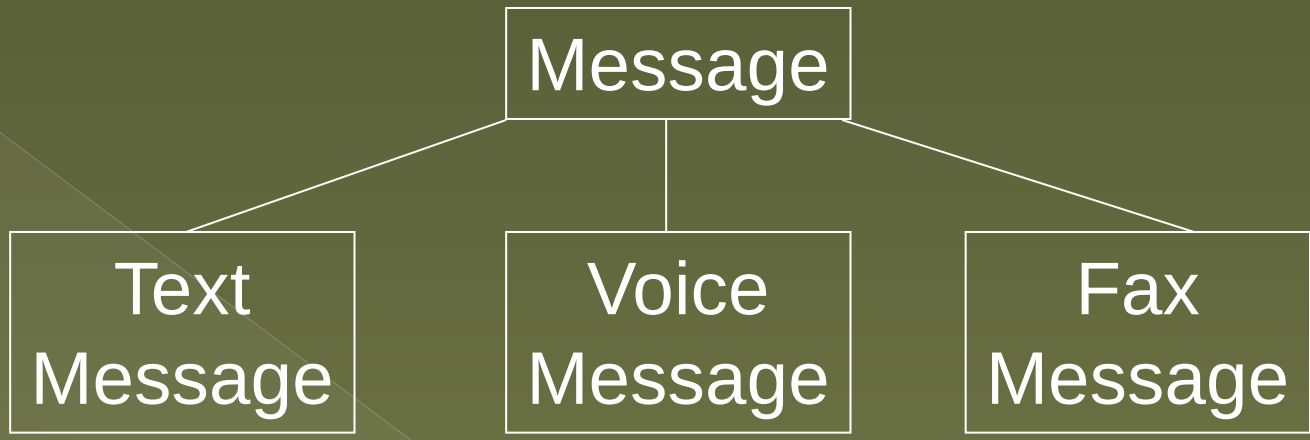
```
//This is the OO (= good) way of computing areas
class Test {
  public static void main(String[] args) {

ClosedCurve[] objects = {
        new Triangle(5,5,5),

        new Square(3),

        new Circle(3)};

//compute areas

for(ClosedCurve cc : objects) {

        System.out.println(cc.compute Area());


        }
    }
                }
```

Points to Observe

- No testing of types is required to access the computeArea() method
- New types of objects (such as Rectangle) can now be introduced by adding new subclasses to ClosedCurve. The only change to the code that is needed is inclusion of new instances in the ClosedCurve[] array, when it is initialized.
- This is an example of the ***Open-Closed Principle***: a well-designed OO program is open to extension but closed to modification.

Other Example






```
public abstract class Message {
  public abstract void display();
}
```

```
public abstract class AProduct
{
   private String productid;

   public void setProductId (String id){
    productid=id;
   }
   public String getProductId (){
    return productid;
   }

   public abstract double getPrice();
}

public class Bicycle extends AProduct
{
   public  double getPrice(){
     return 230.45;
   }
}
```

Not possible

```
AProduct mybycicle = new AProduct();

Bicycle mybycicle = new Bicycle();
```

Disabling Inheritance(final class and final method)

- The final class cannot be extended:

```
final class Math {
   ...
}
```

- The final variable is a constant. Value can not be altered.

```
final static double PI = 3.14159;
```

- The final method cannot be overridden by its subclasses.

- Benefits : **Security** – does not inherit by others.

Example

```
final class A {
final void meth1()
{
System.out.println("This is a final method.");
}
}
class B extends A  // Error
{
void meth1() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Summary about Abstraction

- A class may be declared abstract even if it does not have an abstract method
- Cannot create an object of an abstract class. However , you can declare a variable of the abstract class type and call methods using it.
- An abstract class cannot be declared as final.
- An abstract class should not declare constructors as private.
- An abstract method can not be declared static and private.

Tell the valid definition : e,f

Invalid definition : a b,c,d

```
class A {
  abstract void unfinished() {
  }
}
```

(a)

```
public class abstract A {
  abstract void unfinished();
}
```

(b)

```
class A {
  abstract void unfinished();
}
```

(c)

```
abstract class A {
  protected void unfinished();
}
```

(d)

```
abstract class A {
  abstract void unfinished();
}
```

(e)

```
abstract class A {
  abstract int unfinished();
}
```

(f)

Interface

- A Java *interface* is a way of describing what classes should do, without specifying how they should do it.
- Instance variables and methods can occur. Methods should not have body.
- Interfaces can't have constructors because we can't instantiate them.
- By default any attribute of interface is **public**, **static** and **final**, so we don't need to provide access modifiers to the attributes but if we do, compiler doesn't complain about it either.
- By default interface methods are implicitly **abstract** and **public**, it makes total sense because the method don't have body and so that subclasses can provide the method implementation.

Interface

- An interface is declared using the ***interface*** keyword.
- A class that implements an interface uses the ***implements*** keyword rather than the *extends* keyword.
- Interface can have multiple methods.

Syntax to create an interface

<modifier> interface <interface name>
//public/default and abstract
{
//interface fields(By default Public, Static and Final)
//interface methods(By default Public and Abstract);
}

```
interface Speaker {
    public void speak( );
}
```

- Can implement more than one interface.

Syntax:

MyClass implements Intface1, Intface2, Intface3

- Can also extend *and* implement.

Syntax:

MyClass extends SuperClass implements Intface1, Intface2

```
final class Triangle  implements ClosedCurve {
        final double base;
        final double height;
Triangle(double base, double height){
                this.base = base;
               this.height=height;

        }
double computeArea() {
                return (0.5 * base * height);
        }
}

final class Square implements ClosedCurve {
        final double side;
        Square(double side) {
                this.side = side;
        }
        double computeArea() {
                return(side*side);
        }
}

final class Circle implements ClosedCurve {
        final double radius;
        Circle(double radius) {
                this.radius = radius;
        }
        double computeArea() {
                return (Math.PI * radius * radius);
                        }
                }
```

```
public interface ClosedCurve {
        double computeArea();
}

class Test {
  public static void main(String[] args) {

ClosedCurve[] objects = {

        new Triangle(5,5),

        new Square(3),

        new Circle(3)};

        //compute areas

for(ClosedCurve cc : objects) {

System.out.println(cc.computeArea());

        }
    }
                }
```

Note : This example illustrates the fact that a class that implements an interface may be as the type of that interface.

Interfaces in the Java Library: Comparable

- The `compareTo` method in `String`, `Integer`, `Double`, etc, is in every case an implementation of Java's `Comparable` interface. For `String`s, the interface looks like this:

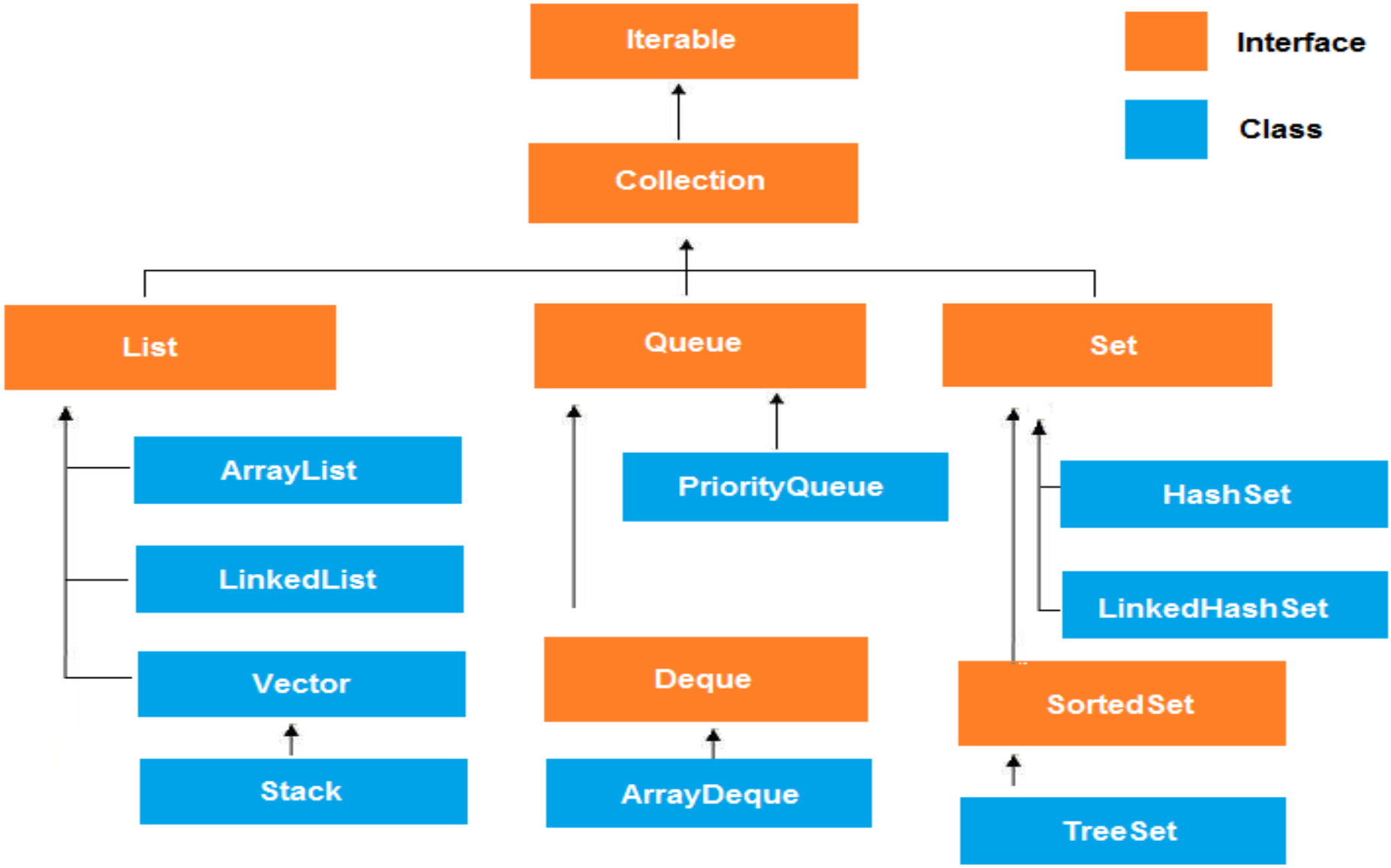
```
interface Comparable<String> {
        public int compareTo(String s);
}
```

In the Java library, `Comparable` is defined *generically*, for any possible type, like this:

```
interface Comparable<T> {
        public int compareTo(T s);
}
```

Note: Generic types will be discussed more in Lesson 8

Predefined Interface Collection



Interfaces in Software Development

- *Interfaces* provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. But more than one *Interfaces can be implemented*.

New Interface Features in Java 8

- Before Java 8, as we have seen, none of the methods in an interface had a method body; all were unimplemented.

- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.
  - A default method is a fully implemented method within an interface, whose declaration begins with the keyword default
  - A static method is a fully implemented method within an interface, having the same characteristics as any static method in a class.

Example – default methods

```
public interface NameAddress {
        //abstract methods
        String getFirstName();
        String getLastName();
        String getStreet();
        String getCity();
        String getState();
      // implemented Methods
        default String getFullName() {
                return getFirstName() + " " + getLastName();
        }
        default String getFullAddress() {
                return getStreet() + "\n" + getCity() + ", " + getState();
        }
}
```

Rules About Default Methods confliction

- If a class implements an interface with a default method, that class inherits the default method (or can override it). However, there are rules to handle the cases in which two interfaces have the same method, or one interface and a superclass have the same method:

- Rule 1 : *Superclass vs Interface- Superclass wins!*
  - When a class extends a superclass and also implements an interface, and both super class and interface have a method with the same name, the superclass implementation wins – this is the version that is inherited by the class.
- Rule 2 : *Interface vs Interface* – clash!
  - If the inherited default method comes from multiple super interfaces , the method from the most specific super interface is inherited by the class.
  - See Democode : Package conflicts

Conflicts

```
Interface I1{
default void display(){
System.out.println("Interface I1");
}
}
```

```
class C1{
void display(){
System.out.println("Class C1");
}
}
```

```
Interface I2{
default void display(){
System.out.println("Interface I2");
}
}
```

```
// Interface conflicts
Class Test implements  I1, I2{
}
```

```
// Super class & Interface conflicts
Class Test extends C1 implements  I1{
}
```

Example – static methods

```
interface FPP
{
  static String CourseID(){
      return "CS390";
}
}

public class StaticMethodsDemo {
public static void main(String args[])
{
System.out.println("Course ID for FPP is : " + FPP.CourseID());
}
}
```

Interfaces in Java 7 and Java 8

Interview Question: What is the difference between an abstract class and an interface?

- **Answer from the perspective of Java 7 (and before)**
  - Abstract classes may have fully implemented and unimplemented methods, but interfaces may not
  - Abstract classes may contain static methods while interfaces may not
  - Abstract classes may have instance variables of any kind, whereas interfaces can have only public static final variables
  - All methods in an interface are public and abstract, but abstract classes may have implemented methods of any visibility and abstract methods that are public or protected or default.
- **Answer from the perspective of Java 8 (and after)**
  - Abstract classes may have fully implemented methods; interfaces may also have implemented methods, but they must bear the keyword “default” /”static”
  - Abstract classes may have instance variables of any kind, whereas interfaces can have only public final static variables
  - All methods in an interface are public, but abstract classes may have implemented methods of any visibility and abstract methods that are public or protected or default.

Functional Interfaces

- Whenever an interface has *just one abstract method*, the interface is called a *functional interface.* The reason for the terminology is that, since there is just one abstract method, implementations of a functional interface behave like a *function.*
- Comparable is an example of a functional interface
- Functional interfaces have become an important aspect of Java since Java SE 8 because implementations can be represented using *lambda expressions* (see Lesson 7)

User-Defined Functional Interfaces

- Defining your own functional interface is easy, but if you want to use it in the context of new Java 8 features, you have to prevent other developers from accidentally adding methods to your interface. This can be done by using the @FunctionalInterface annotation.

```
@FunctionalInterface
public interface MyFunctional {
    void myMethod(String t);

    //if uncommented, there will be a compiler error
    //int anotherMethod(int x);
}
```

MAIN POINT

- Interfaces are used in Java to specify publicly available services in the form of method declarations.
- A class that implements such an interface must make each of the methods operational.
- Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance.
- The concept of an interface is analogous to the creation itself – the creation may be viewed as an "interface" to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

DAY - 3

Introduction to the Reflection Library

- Java have the API library to work with reflection is from **java.lang.reflect.*;**
- Reflection in Java allows an object to
  - determine information about other objects at runtime (such as attributes, methods, constructors)
  - instantiate another object given just the name of the class (and names or types of the parameters passed to the constructor, if any)
  - call a function based only on the name of the function, the class to which it belongs, and the names or types of the function arguments, if any
- For this course, we will see how Reflection can work in conjunction with polymorphism. We will see how techniques of Reflection give us more flexibility in creating polymorphic code.

Example : ReflectionDemo.java

Challenge: Accessing Types Through Reflection

- In the ClosedCurve example, how can we make it so that not only is each area printed out in the for each loop, but also the *type* of closed curve, as in the following:

The area of this Triangle is 12.5

The area of this Square is 9.0

The area of this Circle is 28.274

- We do not want to test the type of each object in the array – this would undermine our implementation of polymorphism. How can we output the type of each object in a generic way?

```
public class Test {
        public static void main(String[] args) {

        ClosedCurve[] objects = {new Triangle(5,5), new Square(3), new Circle(3)};
                //compute areas
                for(ClosedCurve cc : objects) {
                        System.out.println(cc.computeArea());
                }
        }
}
```

- Application: These features of the Class class allow us to solve the Challenge:

```
public class TestSecond {

        public static void main(String[] args) {

        ClosedCurve[] objects = {new Triangle(10,9,6),  new Square(3),
                                        new Circle(3)};
                //compute areas
        for(ClosedCurve cc : objects) {
                String nameOfCurve =
cc.getClass().getSimpleName());

                System.out.println("The area of this "+nameOfCurve+ "
is "+                           cc.computeArea());

                }
        }
```

Challenge: Dynamic Construction with Parameters

In modern-day enterprise Java frameworks (like Spring), reflection is used to “wire together” Java classes in the background so that unnecessary dependencies between classes are eliminated.

Spring uses an XML configuration file in which the names of classes are recorded, along with information about the relationships between the classes. This configuration is then used at startup – *using Reflection* – to create instances of the main classes for the application with dependencies realized exactly as intended.

See : ReflectionDemo.java

MAIN POINT

The classes in the Java reflection package can be used to construct an instance of a class.

Likewise, reflection on the infinite creative power of consciousness reveals the truth of every thing and gives rise to the creation of any object.

The Object Class & Methods

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- *Singly-rooted*. Every Java class belongs to one large inheritance hierarchy in which Object is at the top. No explicit mention of "extending" Object needs to be made in your code – it is already understood by the compiler and JVM.
- Every class has access to the following methods (and others that we will not cover here):
  - `public String toString()`
  - `public boolean equals(Object o)`
  - `public int hashCode()`
  - `protected Object clone() throws CloneNotSupportedException`

Refer : Additional Reading for detailed information

The `toString` Method

1. If a class does not override the default implementation of `toString` given in the `Object` class, it produces output like the following:

```
public static void main(String[] args) {
    System.out.println(new Object());
    System.out.println(new StoreDirectory(null));

}

//output
java.lang.Object@18d107f
scope.more.StoreDirectory@ad3ba4
```

This is a concatenation of the fully qualified class name with the hexadecimal version of the "hash code" of the object (we will discuss hash codes later in this set of slides)

2. Most Java API classes override this default implementation of `toString`. The purpose of the method is to provide a (readable) `String` representation (which can be logged or printed to the console) of the state of an object.

Example from the Exercises:

```
// the Account object has this implementation of        //
 toString
public String toString(){
     String newline =
        System.getProperty("line.separator");
     String ret =
        "Account type: " + acctType + newline +
              "Current bal:  " + balance;
     return ret;
}
```

Best Practice. For every significant class you create, override the `toString` method.

3. `toString()` is automatically called when you pass an object to `System.out.println` or include it in the formation of a `String`

4. Examples:

```
Account acct = . . . //populate an AccountString
output = "The account: " + acct;
```

```
Account acct = . . . // populate an Account
System.out.println(acct);
```

5. toString for arrays – sample usage

Suppose we have the array

```
String[] people = {"Bob", "Harry", "Sally"};
```

- Wrong way to form a string from an array

```
people.toString()
  //output:  [Ljava.lang.String;@19e0bfd
```

- Right way to form a string from an array

```
Arrays.toString(people)
    //output:   [Bob, Harry, Sally]
```

The Object Class – equals()

- Implementation in Object class:

ob1.equals(ob2)
if and only if ob1 == ob2
if and only if references point to the same object

- Using the '==' operator to compare objects is usually not what we intend (though for comparison of *primitives*, it is just what is needed). For comparing objects, the equals method should (usually) be overridden to compare the *states* of the objects.

Example

```
//an overriding equals method in the Person class
@Override
public boolean equals(Object aPerson) {
        if(aPerson == null) return false;
        if(!(aPerson instanceof Person)) return false;
        Person p = (Person)aPerson;
        boolean isEqual = this.name.equals(p.name)
        return isEqual;
}
```

Handling equals() in Inherited Classes

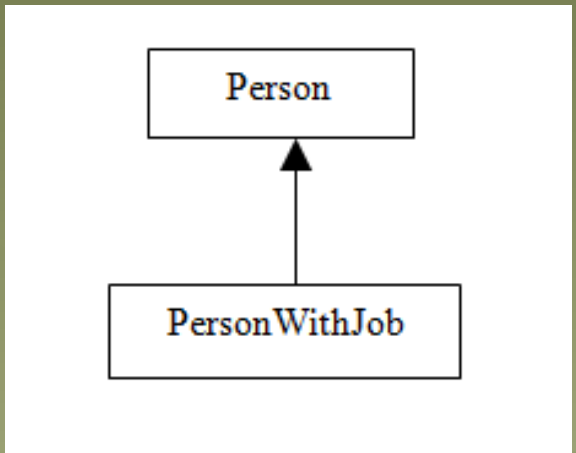
If a class has no inheritance check the equals() using either getClass() or instanceof

Checking of equals() using getClass() is called same class strategy.

Checking of equals() using instanceof is called instance of strategy

Refer Examples from equals package(case1,case2,case3)

Example: Add a subclass PersonWithJob to Person:



Case 1 : instsnceof Strategy

```
class Person {
	private String name;
	Person(String n) {
		name = n;
	}
	pubic String getName() {
		return name;
	}
	@Override
	public boolean equals(Object aPerson) {
		if(aPerson == null) return false;
		if(!aPerson instanceof Person) return false;
		Person p = (Person)aPerson;
		boolean isEqual = this.name.equals(p.name)
		return isEqual;
	}
}
class PersonWithJob extends Person {
	private double salary;
	PersonWithJob(String n, double s) {
		super(n);
		salary = s;
	}
}
```

The equals() method is inherited by PersonWithJob in this implementation. So objects of type PersonWithJob are compared only on the basis of the name field.

Example:

```
PersonWithJob joe1 = new PersonWithJob(“Joe”, 100000);
PersonWithJob joe2 = new PersonWithJob(“Joe”, 50000);
boolean areTheyEqual = joe1.equals(joe2);
//areTheyEqual
      == true
```

Best Practices: If, in your code, this kind of situation does not present a problem – if it is OK to inherit `equals()` in this way – then the implementation given here is optimal. This is called the *instanceof strategy for overriding equals*

Best practice in the case where subclasses need to have their own form of equals is more complicated (discussed in next slide)

What Happens When Subclasses Need Their Own Form of equals()

Case 2 : same class Strategy

Example. Provide `PersonWithJob` its own `equals` method.

```
//an overriding equals method in the PersonWithJob class
@Override
public boolean equals(Object ob) {
        if(ob == null) return false;
        if(this.getClass() != ob.getClass())return false;
        PersonWithJob p = (PersonWithJob)withJob;
        boolean isEqual= getName().equals(p.getName()) &&
                                this.salary == p.salary;
        return isEqual;
}
```

This creates a serious problem, called *asymmetry* (violates contract for equality)

```
Person p = new Person(“Joe”);
PersonWithJob withJob = new PersonWithJob(“Joe”,
                                          100000);
//true – using Person’s equals()
theyreEqual1 = p.equals(withJob);

//false – using PersonWithJob’s equals()
theyreEqual2 = withJob.equals(p);
```

Best Practice When Using Same Classes Strategy

The example shows that whenever the same classes strategy is used to provide separate equals methods for classes B and A, where B is a subclass of A, then we should prevent the possibility of creating a subclass of B to prevent the introduction of a corrupted equals method.

Best Practice – Same Classes Strategy. If B is a subclass of A and each class has its own equals method, implemented using the same classes strategy, then the class B should be declared `final` to prevent the introduction of an asymmetric definition of equals in any future subclass of B.

Question. What if we don't wish to make B `final`?

Using Composition Instead of Inheritance

Case 3 : Composition instead of Inheritance

Even when a potential subclass satisfies IS-A criterion for inheritance, we might not choose to use inheritance, as long as we do not need the subclass for polymorphism.

The discussion above is one such case: Whenever classes B and A, where B is a subclass of A, require different `equals` methods, using composition instead of inheritance is a good strategy, and if making the class B `final` is not an option, it is the only safe way to handle `equals`.

Example: *Implementing PersonWithJob using Composition instead of Inheritance*

Summary of Best Practices for Overriding Equals In the Context of Inheritance

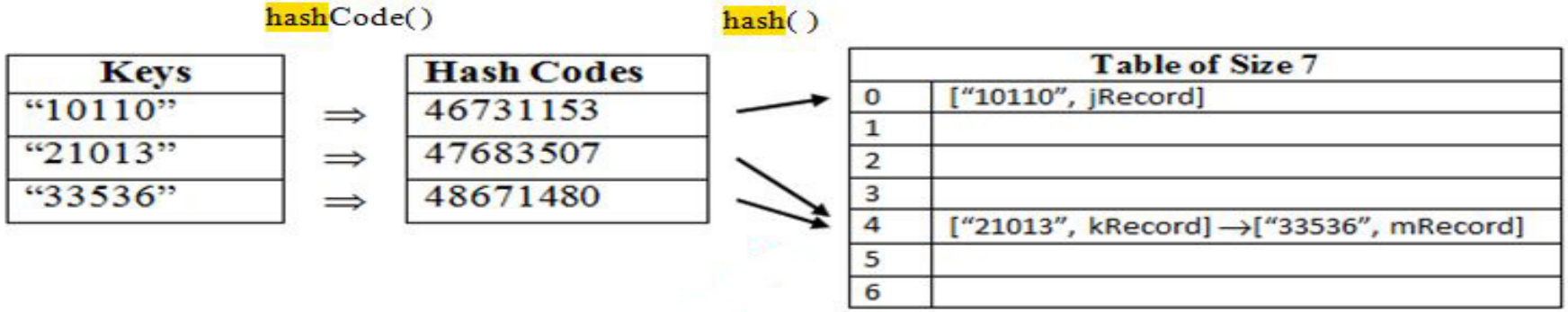
Suppose B is a subclass of A.

- If it is acceptable for B to use the same equals method as used in A, then the best strategy is the *instanceof strategy*
- If *two different equals methods* are required, two strategies are possible
- Use composition instead of inheritance – this will always work as long as the inheritance relationship between B and A is not needed (e.g. for polymorphism)
- Use the *same classes* strategy, but declare subclass B to be final

Overriding hashCode()

When objects of any kind (Integers, Strings, chars, or any others) are used as keys in a hashtable (discussed in Lesson 11), Java will use the hashCode() method available in the class to transform each key into a small integer, serving as an index in an underlying array.

User's View of Hashtable	
Key (= Employee ID)	Value (= Record)
"10110"	jRecord
"21013"	kRecord
"33536"	mRecord



(continued)

- For this mechanism to function correctly, the following rule must be followed:

HashCode Rules

1. *Whenever equals() is overridden in a class, hashCode() must also be overridden*
2. *The hashCode method must take into account the same fields as those that are referenced in the overriding equals method.*

- The reason for this rule more details about hashing will be discussed in Lesson 11.
- Example : ObjectMethodsDemo.java

Creating a Hash Value from Object Data (From Effective Java, 2nd Ed.)

- You are trying to define a hash value for each instance variable of a class. Suppose f is such an instance variable.
  - If f is boolean, compute (f ? 1 : 0)
  - If f is a byte, char, short, or int, compute (int) f.
  - If f is a long, compute (int) (f ^ (f >>> 32))
  - If f is a float, compute Float.floatToIntBits(f)
  - If f is a double, compute Double.doubleToLongBits(f) which produces a long f1, then return (int) (f1 ^ (f1 >>> 32))
  - If f is an object, compute f.hashCode()

The clone() Method

- Java does not provide an automatic mechanism to clone (make a copy) an object.
- Recall that when you assign a reference variable to another reference variable, only the reference of the object is copied, not the content of the object.
- Cloning an object means copying the content of the object bit by bit.
- Steps to implement clone()
  - implement the Cloneable interface
  - override the clone method with public/protected access privileges from the java.lang.Object class.
  - call super.clone()
  - Handle CloneNotSupported Exception.
- Cloneable is an interface declared in the java.lang package.

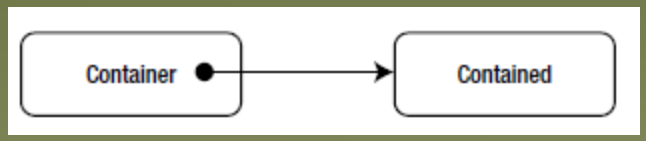
The declaration of the clone() method in the Object class is as follows:

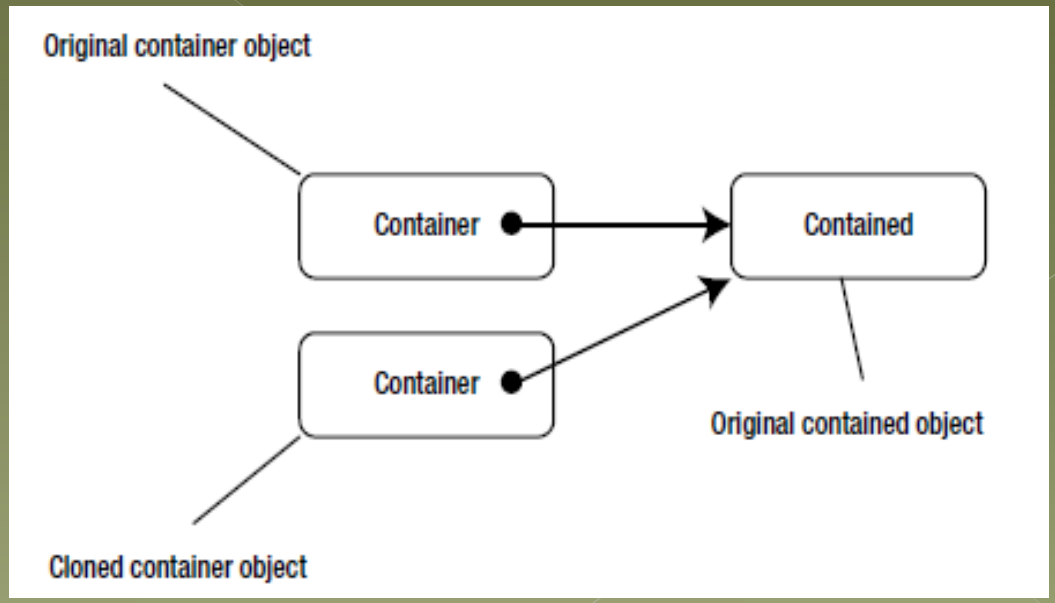
protected Object clone() throws CloneNotSupportedException

Examples : lesson7democode.cloneshadow; **package lesson7democode.clonedeep;**

Cloning by Shallow copy

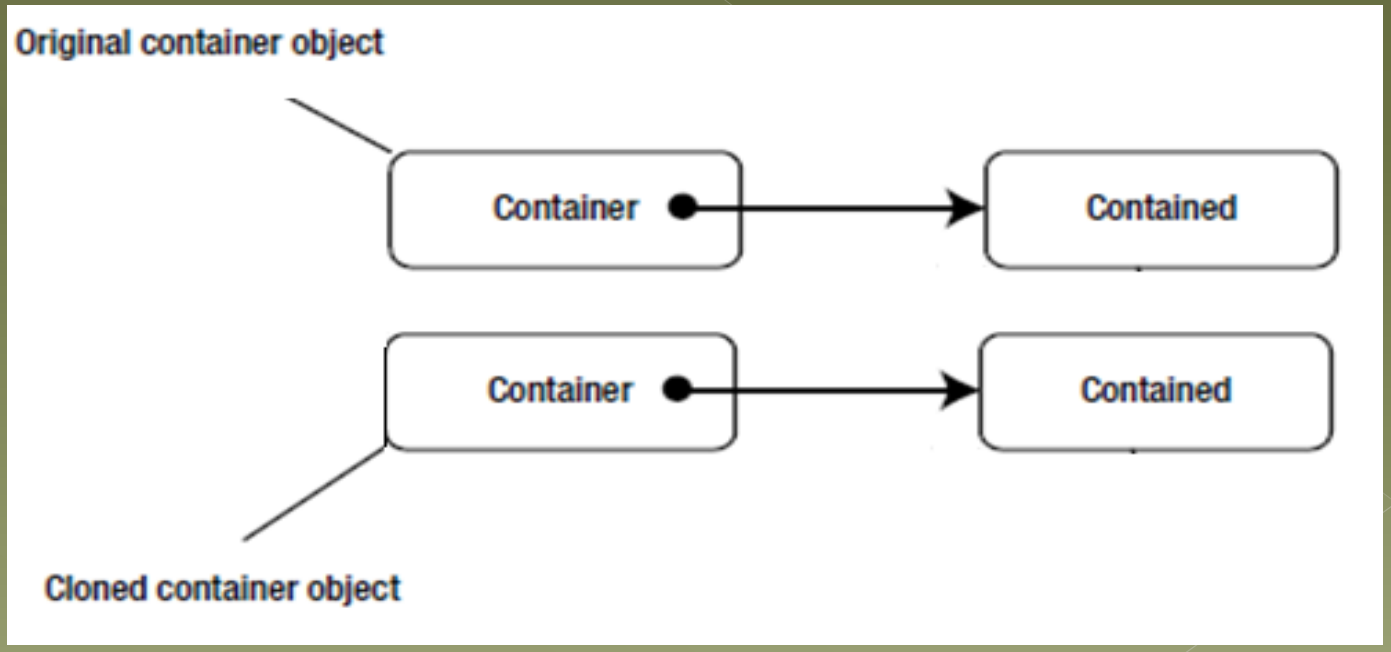
- An object may be composed of another object. In such cases, two objects exist in memory separately—a contained object and a container object.
- The container object stores the reference of the contained object.
- When you clone the container object, the reference of the contained object is cloned. After cloning is performed, there are two copies of the container object; both of them have references to the same contained object.
- This is called a shallow cloning because references are copied, not the objects.





Cloning by Deep copy

- When the contained objects are copied rather than their references during cloning of a compound object, it is called deep cloning.
- You must clone all the objects referenced by all reference variables of an object to get a deep cloning.



Class Relationships

- Object of one class may be related to the objects of another class is the following ways.
  - Is-A Relationship (Inheritance)
  - Has-A Relationship(Aggregation / Composition)
- A class can have reference to the object of other class as member is Has-A relationship. One form of software reuse is composition.
- Eg : Date class
- Demo : CompositionDemo.java

PRACTICE