CENG 1004
Introduction to Object Oriented Programming

Spring 2016

WEEK 7

Today's Topics

- Lecture 6 Review
- Generics

Lecture 6 Review

Casting

- It is always possible to convert a subclass to a superclass. For this reason, explicit casting can be omitted. For example,
  - `Circle c1 = new Circle(5);`
  - `Object s = c1;`

is equivalent to

  - `Object s = (Object)c1;`

- Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.
  - `Circle c2 = (Circle) s;`

instanceof

```
Rectangle r = new Rectangle(5, 10);
Circle c = new Circle(5);
Object s = c;
if(s instanceof Rectangle)
      ((Rect)s).changeWidth(4);
```

- syntax: **expression instanceof ClassName**

Abstract Shape

```
public abstract class Shape {

    public abstract double area();

    public abstract double perimeter();

}
```

Circle extends Shape

```
public class Circle extends Shape{
    private double radius;
    ...
    public double area(){
        return Math.PI * Math.pow(radius, 2);
    }
}
```

Rectangle extends Shape

```
public class Rectangle extends Shape{

    double width;
    double height;
    ...
    public double area(){
          return height * width;
    }
}
```

Drawing (Version 3)

```
public class DrawingV3 {

    ArrayList<Shape> shapes = new ArrayList<Shape>();

    public void addShape(Shape shape){
        shapes.add(shape);
    }

    public double calculateTotalArea(){
        double totalArea = 0;
        for (Shape shape : shapes){
            totalArea += shape.area();
        }
        return totalArea;
    }
}
```

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next

Polymorphism

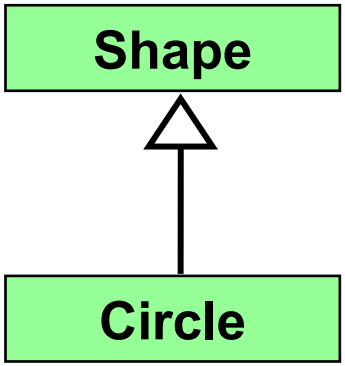
- Suppose we create the following reference variable:

```
Shape shape;
```

- Java allows this reference to point to an `Shape` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

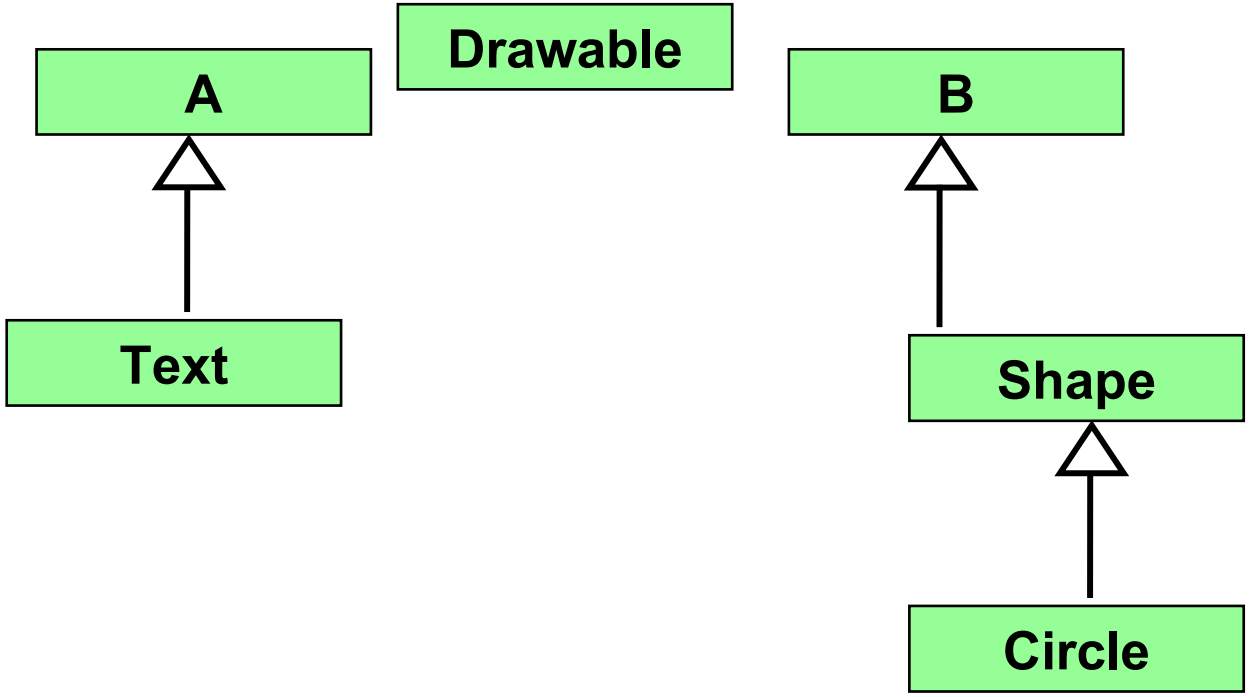
References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Shape` class is used to derive a class called `Circle`, then a `Shape` reference could be used to point to a `Circle` object




```
Shape shape;
shape = new Circle(5);
```

Assume Text and Shape have already superclasses



- Multiple Inheritance is not allowed in Java

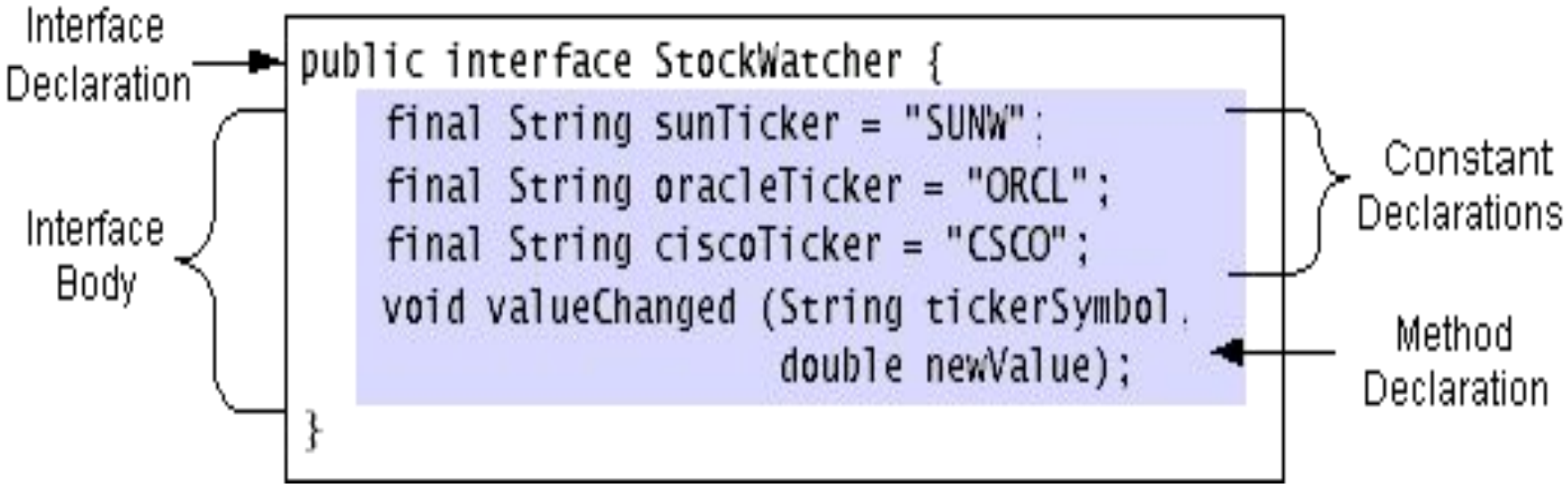
Interface

- An interface is a named collection of method definitions and constants ONLY.
- 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.
- A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behaviors.

Interface and Abstract Classes

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Defining Interfaces



Interface Body

- The interface body contains method declarations for **ALL** the methods included in the interface.
- A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it.
- All methods declared in an interface are implicitly public and abstract.

Drawable Interface

```
public interface Drawable {

    public void draw();

}
```

Drawable Text

```
public class Text  implements Drawable{

    private String text;

    public Text(String text){
        this.text = text;
    }

    public void draw(){
        //to be imlemented
    }
}
```

Generics

Generics

- Some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on

- Generics add stability to your code by making more of your bugs detectable at compile time.

Why Use Generics

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
  - Stronger type checks at compile time.
  - Elimination of casts.
  - Enabling programmers to implement generic algorithms.

Generics Example

- The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Generic Types

- A generic type is a generic class or interface that is parameterized over types.

- The following Box class will be modified to demonstrate the concept.

```
public class Box {
    private Object object;

    public void set(Object object){
              this.object = object;
    }
    public Object get() {
              return object;
    }
}
```

A Generic Version of the Box Class

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) {
       this.t = t;
    }
    public T get() {
       return t;
    }
}
```

Generic Class Definition

- A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn>{
   ...
 }
```

- This same technique can be applied to create generic interfaces

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
- The most commonly used type parameter names are:

  E - Element (used extensively by the Java Collections Framework)

  K - Key

  N - Number

  T - Type

  V - Value

  S,U,V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

- To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces **T** with some concrete value, such as **Integer**:

```
Box<Integer> integerBox;
```

- Similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

Invoking and Instantiating a Generic Type

- Like any other variable declaration, this code does not actually create a new Box object.

- It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

- To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

  Box<Integer> integerBox = new Box<Integer>();

The Diamond

- You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context.

```
Box<Integer> integerBox = new Box<>();
```

- This pair of angle brackets, <>, is informally called the diamond.

Multiple Type Parameters

- As mentioned previously, a generic class can have multiple type parameters.

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

Multiple Type Parameters

```
public class PairImpl<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public PairImpl(K key, V value) {
      this.key = key;
      this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Multiple Type Parameters

- The following statements create two instantiations of the PairImpl class:

```
Pair<String, Integer> p1 = new PairImpl<String, Integer>("Even", 8);

Pair<String, String>  p2 = new PairImpl<String, String>("hello",
   "world");
```

Multiple Type Parameters

- You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>).

```
Pair<String, Box<Integer>> p = new PairImpl<>("primes", new
   Box<Integer>(...));
```

Raw Types

- A raw type is the name of a generic class or interface without any type arguments.

- If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Raw Types

- When using raw types, you essentially get pre-generics behavior — a Box gives you Objects.
- For backward compatibility, assigning a parameterized type to its raw type is allowed.
- You should avoid using raw types.

Generic Methods

- Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1,
       Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

Generic Methods

- The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

- The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
boolean same = Util.compare(p1, p2);
```

- This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

Bounded Type Parameters

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type

- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

Bounded Type Parameters

- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, which in this example is Number.

```
public <U extends Number> void inspect(U u){

  System.out.println("U: " + u.getClass().getName());

}
```

Generic Methods and Bounded Type Parameters

```
public static <T> int countGreaterThan(T[] anArray, T
   elem) {
     int count = 0;
     for (T e : anArray)
         if (e > elem)  // compiler error
             ++count;
     return count;
}
```

- Does not compile because the greater than operator (>) applies only to primitive types such as short, int, double, long, float, byte, and char.

- You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the Comparable<T> interface:

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

Generic Methods and Bounded Type Parameters

The resulting code will be:

```
public static <T extends Comparable<T>> int
   countGreaterThan(T[] anArray, T elem) {

    int count = 0;

    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;

    return count;
}
```

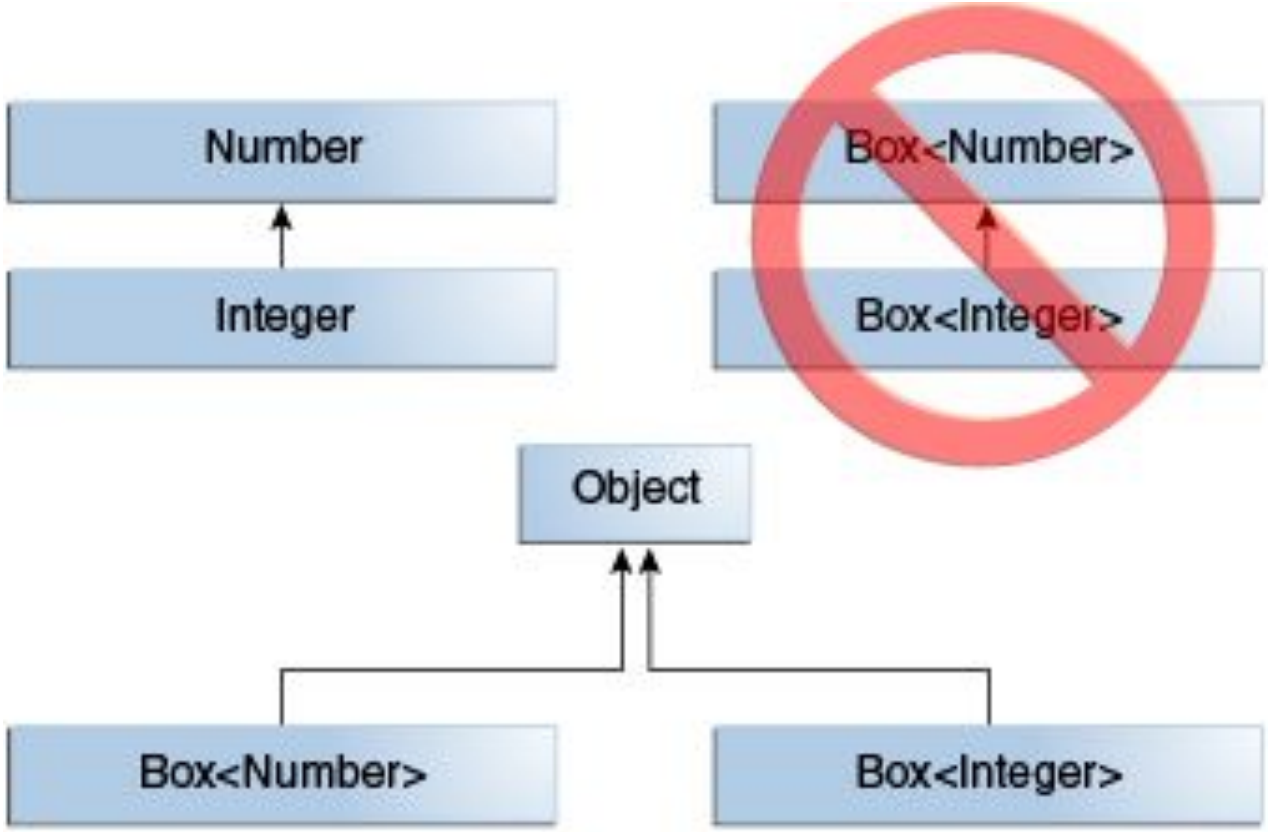
Multiple Bounds

- A type parameter can have multiple bounds:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

Generics, Inheritance, and Subtypes



Generics, Inheritance, and Subtypes

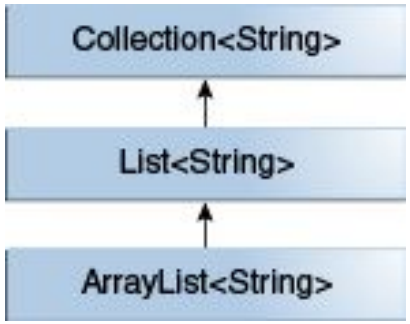
- Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

- Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect?

- The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

Generic Classes and Subtyping



- `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`.
- So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`.
- So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

Generic Classes and Subtyping

Is the following possible?

```
List<String> listStrings = new ArrayList<String>();
List<Object> listObjects = listStrings;
```

Well, we know that the following is of course fine:

```
String str = "hello";
Object obj = str;
```

Answer is: NO (compilation error)

This comes to avoid the following:

```
listObjects.add(7);
String str = listStrings.get(0); // wd've been run-time error
```


47

Wildcards

Suppose we want to implement the following function:

```
void printCollection(Collection col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

But we want to do it in a "generic" way, so we write:

```
void printCollection(Collection<Object> col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Can get ONLY collection of Objects

(go one slide back for explanation)

Cannot support Collection<String> Collection<Float> etc.

What's wrong with the 2nd implementation?

48

Wildcards

The proper way is:

```
void printCollection(Collection<? extends Object> col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Which is the same, for this case, as:

```
void printCollection(Collection<?> col) {
        for(Object obj : col) {
                System.out.println(obj);
        }
}
```

Now we support all type of Collections!

49

Wildcards

One more wildcard example:

```
public interface Map<K,V> {
        …
        void putAll(Map<? extends K, ? extends V> map)
        …
}
```

And another one:

```
public interface Collection<E> {
        …
        void addAll(Collection<? extends E> coll)
        …
}
```


50

Summary for today

- Generics

References

- http://math.hws.edu/javanotes/
- https://docs.oracle.com/javase/tutorial/java/generics/
- http://www2.mta.ac.il/~amirk/java/presentations/03-Collections.ppt
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/20_ch22_Collections.ppt
- https://examples.javacodegeeks.com/core-java/util/comparator/java-comparator-example/