Consolidating Features

Rui S. Moreira

Access control - modifiers

- private:
  variables or methods accessible only to the class
- public:
  variables or methods accessible to all classes
- default (no explicit modifier):
  variables or methods accessible to all classes of the same package
- protected:
  variables or methods accessible to all classes and subclasses of the same package

Invoke overridden methods

```
public class A {
   public float add(float i, float j){
        return i+j;
   }
}

public class B extends A {
   // Modifier of overridden method cannot give less accessibility
   // Overridden method cannot throw more exceptions
   @Override
   public float add(float i, float j){
        return java.math.Math.round(super.add(i, j));
   }
}
```


Invoking overloaded methods/constructors

```
public class Employee {
   String name;
   int salary;
   public Employee (String n, int s){
        this.name=n;
        this.salary=s;
   }
   public Employee (String n){
        this(n, 0);
   }
   public Employee (){
        this("unknown");
   }
   /** Copy constructor */
   public Employee (Employee clone){
        this(clone.name, clone.salary);
   }
}
```

Invoke parent constructors

When creating new objects:

1. Allocate memory for object and init member variables with 0 values (primitive types) or null values (references);
2. Explicitly init member variables;
3. Invoke constructor.

```
public class Employee {
   String name = "unknown";
   public Employee (String n){
        this.name=n;
   }
}

public class Manager extends Employee {
   String department = "";
   public Manager (String n, String d){
        super(n); //Must be first method to be called
        this.department = d;
   }
}
```


Static class variables (global)

```
public class Count {

   // Static variable is similar to "global variable"
   // since it is shared between all the instances of the class
   private static int counter = 0;

   private int serialNumber;

   public Count(){
        counter++;
        this.serialNumber=counter;
   }
}
```

Static class methods (global)

```
public class Count {
   int x;

   // Static method is invoked on the class, i.e.,
   // does not need an object to be invoked
   public static int add(int a, int b){
        // Error - static method cannot access non-static vars
        //x=a+b; return x;
        return a+b;
   }
}

public class CountApp {

   public static void main(String args[]){
        int a=5, b=6;
        System.out.println("Soma "+a+" + "+b+"="+ Count.add(a,b));
   }
}
```


final classes, methods and attributes

```
// Final class: cannot be extended/sub-classed
// e.g. String is final - security reasons
public final class A {

   // Final attribute: constant for all instances
   private final float pi = 3.1415;

   // Final method: not possible to be overridden
   // when calling the method we are sure it was not changed
   public final int add(int a, int b){
        return a+b;
   }
}
```

PS1: static or private methods are automatically final

PS2: java –o (compiles with optimization - invokations are embeded, i.e., static, private and final methods are called directly instead of using a virtual method invokation)

Abstract classes and methods

```
// Abstract class: cannot be instantiated
//(since it has non implemented methods)
public abstract class Drawing {
   // At least one method is not implemented
   public abstract void drawDot(int x, int y);
}
```


Inner classes (Pascal concept)

```
// Inner class (e.g. create event adapters):
// visible only inside the class where declared
// we may choose not to give the class a name (unnamed)
   JFrame f = new JFrame("...");

   //Add a mouse motion listener to the JFrame
   f.addMouseMotionListener(new MouseMotionAdapter() {
            // Define class members and methods here
            public void mouseDragged(MouseEvent mv){
                    //...
            }
       } //End declaration of unnamed/inner class
   );
}
```

Generics

- Generics are used to parameterize a class
  - e.g. suppose we have a class `BoxObject` to store any object inside:

```
public class BoxObject {

    private Object object;

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

However, at compile time, it is not possibly to verify/restrict the type of the inner object.

Generics

- With a generic type T we may parametrize the class and restrict its inner object type

```
public class BoxGeneric<T> {

    // T stands for generic "Type"
    private T t;

    public BoxGeneric(T t) { this.t = t; }

    public void setT(T t) { this.t = t; }

    public T getT() { return t; }
}
```

Generics

- Generic types may also parameterize methods

```
public class BoxGeneric<T> {

    // T stands for generic "Type"
    private T t;

    // Use extends to restrict the generic type to be used
    public <N extends Number> void inspect(N n){
        System.out.println("T: " + t.getClass().getName());
        System.out.println(”N: " + n.getClass().getName());
    }
}
```


Generics

- Generics may parameterize interfaces as well:

```
/** A type variable can be any non-primitive type (class,
 * interface, array type, another type variable).
 * The most commonly used type parameter names are:
 *  E - Element (used extensively by Java Collections),
 *  K - Key,
 *  N - Number,
 *  T - Type,
 *  V - Value,
 *  S, U, etc. - 2nd, 3rd, types
 */
public interface PairKeyValueI<K, V> {
    public K getKey();
    public V getValue();
}
```

Generics

- Use parameterized `PairKeyValueI` interface on a method:

```
public class UtilGeneric {

  /** Parameterize method with K and V for typifying the generic method parameters */
  public static <K, V> boolean compare(PairKeyValueI<K, V> p1, PairKeyValueI<K, V> p2) {
    return (p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue()));
  }

  //...
}
```


Generics

- Bound a generic type T to be able to compare any `Comparable` objects:

```
public class UtilGeneric {
  //...

  /** The operator greater than (>) applies only to primitive types, hence,
   *  we have a compile-time error below - see solution on next slide */
  public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray) {
      if (e > elem){ // COMPILE ERROR!!
        ++count;
      }
    }
    return count;
  }
}
```

Generics

- Assuming we already have the generic `Comparable` interface:

```
/** The interface Comparable is already parameterized by <T> */
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- Then we may use it to bound the generic type <T>:

```
public class UtilGeneric {
  //...
  /** Use a type parameter bounded by Comparable, which in turn is also parameterized by
   * generic type <T> */
  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;
  }
}
```


Generics

- We may also use the generic PairKeyValueI on a class:

```
public class OrderedPair<K, V> implements PairKeyValueI<K, V> {
    private K key;
    private V value;

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

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey() { return key; }
    public V getValue() { return value; }

    //...
}
```

Generics

```
public class OrderedPair<K, V> implements PairKeyValueI<K, V> {
  //...
  public static void main(String[] args) {
    PairKeyValueI<String, Integer> p1 = new OrderedPair<>("Even", 18);
    PairKeyValueI<String, String> p2 = new OrderedPair<>("Name", "Pedro");
    System.out.println("main(): key = " + p1.getKey());
    System.out.println("main(): value = " + p2.getValue());

    //We may substitute a type parameter (cf. K or V) with a parameterized type
    //(e.g., List<String>, BoxGeneric<Integer>)
    OrderedPair<String, BoxGeneric<Integer>> p3 =
          new OrderedPair<>(”Box Int", new BoxGeneric(18));
    System.out.println("BoxGenericObject – main(): Value = " + p3.getValue().get());

    PairKeyValueI<String, Integer> p4 = new OrderedPair<>("Even", 8);
    PairKeyValueI<String, Integer> p5 = new OrderedPair<>("Even", 8);

    System.out.println("main(): same1=" + UtilGeneric.compare<String, Integer>(p4, p5));
    //We may omit diamond operator
    System.out.println("main(): same2 = " + UtilGeneric.compare(p1, p5););
  }
}
```


Generics

- We may use **Multiple Bounds**, i.e., several extended types (the first must be a class and the others interfaces):

```
public class NaturalNumber<T extends Integer & Comparable<Integer>> {
    private T n;
    public NaturalNumber(T...elements){
        n = elements[0];
    }
    public NaturalNumber(T n) { this.n = n; }
    public T getN() { return n; }
    public void setN(T n) { this.n = n; }

    //This method invokes the intValue() defined in  Integer class
    public boolean isEven() { return (n % 2 == 0); }

    public static void main(String[] args) {
       NaturalNumber<Integer> nn1 = new NaturalNumber(10);
    }
}
```

Generics with wildcards

- We may use an upper bounded wildcard to relax the restrictions on a variable:

```
/**
 * Define a method that works on List<Integer>, List<Double>, List<Number>,
 * by using an upper bounded wildcard, i.e. List<? extends Number>.
 *   - List<Number>: more restrictive because it matches a list of type
 *     Number only, whereas the latter
 *   - List<? extends Number>: matches list of type Number or any of its
 * subclasses (cf. Integer, Double, etc.).
 */
public class UtilGeneric {
  //...

  public static String process(List<? extends Number> list) {
    String s = "";
    for (Number e : list) { s += e.intValue(); }
    return s;
  }
}
```


Generics with Upper bounded wildcards

- We may use an upper bounded wildcard to relax the restrictions on a variable:

```
public class UtilGeneric {
  //...
  public static double sumOfList(List<? extends Number> list) {
    double d = 0.0;
    for (Number n : list) { d += n.doubleValue(); }
    return d;
  }

  public static void main(String[] args) {
    List<Integer> li = Arrays.asList(1, 2, 3);
    System.out.println("sum = " + sumOfList(li)); //Prints: sum = 6

    List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
    System.out.println("sum = " + sumOfList(ld)); //Prints: sum 7.0
  }
}
```

Generics with Unbounded Wildcards

- Unbounded wildcard (?) for managing unknown types:

```
public class GenericUtil {
  //...
  /** Prints any lists of objects but cannot print lists of Integer,
    * Double, String, etc., because List<Integer>, List<Double>,
    * List<String>, etc. are not subtypes of List<Object>.
    */
  public static void printList(List<Object> list) {
    for (Object elem : list){ System.out.println(elem + " "); } }
  }

  /** A generic print method may use an unbounded wildcard. For any
   * concrete type A, List<A> is a subtype of List<?>.
   */
  public static void printList(List<?> list) {
    for (Object elem: list) { System.out.print(elem + " "); }
  }
}
```


Generics with Unbounded Wildcards

- Unbounded wildcard (?) for managing unknown types:

```
public class GenericUtil {
  //...

  public static void main(String[] args) {
    //...
    List<Integer> li = Arrays.asList(1, 2, 3);
    List<String>  ls = Arrays.asList("one", "two", "three");
    GenericUtil.printList(li);
    GenericUtil.printList(ls);
  }
}
```

Generics with Lower bounded wildcards

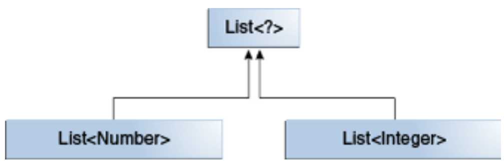
- We may use a lower bounded wildcard to restricts the unknown type to be a specific type or a super type of that type:

```
/**
 * Write a method that puts integers into a list (capable of accepting
 * List<Integer>, List<Number> and List<Object>, i.e., any super types
 * of Integer).
 */
public class GenericUtil {
  //...

  /** List<Integer> is more restrictive than List<? super Integer> because:
   *   - List<Integer>: matches a list of type Integer only
   *   - List<? super Integer>: matches a list of any type that is supertype
   *     of Integer.
   */
  public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) { list.add(i); }
  }
}
```

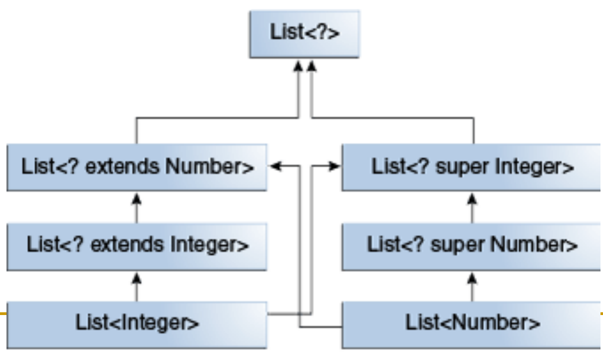

Wildcards and Subtyping

- The common parent of List<Number> and List<Integer> is List<?>



- Since List<? extends Integer> is **subtype** of List<? extends Number>, then we may have:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number> numList = intList;
```



Guidelines for Wildcard Use

- Given methods with “in” and “out” parameters:
  - "in" params may be defined with **upper bounded** wildcards (`extends`).
  - "out" params may be defined with **lower bounded** wildcards (`super`).
  - "in" params accessed through Object class methods use **unbounded wildcard**.
  - When using both ways params ("in" and "out”), then **do not use wildcard**.
- Using a wildcard as a return type should be avoided
  - because it forces programmers using the code to deal with wildcards.

Restrictions on Generics

- Cannot Instantiate Generic Types with Primitives

```
public class Pair<K, V> { /* … */ }
//Compile-time error
Pair<int, char> p = new Pair<>(8, 'a');
```

- Cannot Create Instances of Type Parameters

```
public static <E> void append(List<E> list) {
    E elem = new E(); //Compile-time error
}
```

- Cannot Declare Static Fields Whose Types are Type Parameters

```
public class MobileDevice<T> {
    private static T os; //Compile-time error
}
```

Restrictions on Generics

- Cannot Use Casts or instanceof with Parameterized Types
  - The Java compiler erases all type parameters in generic code, hence cannot verify which generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    //Compile-time error
    if (list instanceof ArrayList<Integer>) {/*...*/}
}
```

  - Instead may use an unbounded wildcard:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  /* OK */ }
}
```

- Cannot Create Arrays of Parameterized Types

```
//Compile-time error
List<Integer>[] aol = new List<Integer>[2];
```


Restrictions on Generics

- Cannot Create, Catch, or Throw Objects of Parameterized Types
  - Class **cannot extend `Throwable`** directly or indirectly:

```
//Compile-time error
class MathException<T> extends Exception {/*...*/}
class QueueFullException<T> extends Throwable {/*...*/}
```

  - Method **cannot catch an instance** of a type parameter:

```
public static <T extends Exception, J> void
execute(List<J> jobs) {
    try {
        for (J job : jobs) // ...
    } catch (T e) {   /* Compile-time error */ }
}
```

  - Method **can use a type parameter** in a throws clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T { /* OK */ }
}
```

Restrictions on Generics

- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type
  - Class **cannot have two overloaded** methods that will have the **same signature** after type erasure:

```
public class Example {
    public void print(Set<String> strSet) { /* ... */ }
    public void print(Set<Integer> intSet) {/* ... */ }
}
```

  - NB: the Java compiler applies type erasure to:
    - Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded (the produced bytecode, therefore, contains only ordinary classes, interfaces, and methods).
    - Insert type casts if necessary to preserve type safety.
    - Generate bridge methods to preserve polymorphism in extended generic types.