Generics

Generic and parameterized types, parameters and return types, interfaces, wild cards, restrictions, wrappers

Generic Types

A **generic type** is a generic class or interface that has a type parameter. Example: class `OrderedPair<T>`.

- T stands for type.
- With collections, it's common to see E for element: `List<E>`

```
public class OrderedPair<T> {
   private T x;
   private T y;

   public OrderedPair(T x, T y) {
      this.x = x;
      this.y = y;
   }

   public T getX() { return x; }
   public T getY() { return y; }
}
```

Parameterized types

You can instantiate a generic type to create a **parameterized type**. That is, you supply an *actual type argument* to the generic type to specify what the *formal type parameter* stands for.

Examples:

- `OrderedPair<String>`
- `OrderedPair<Integer>`
- `OrderedPair<Thermometer> // etc....`

The actual type parameter must be a class or interface.

You can instantiate the parameterized type to create a new object:

```
OrderedPair<String> parents
               = new OrderedPair<String>(“mom”, “dad”);
String myMother = parents.getX();
```

Parameterized types in methods

A formal parameter for a method can be a parameterized type.

A method return type can be a parameterized type.

```
public static void printStrings(OrderedPair<String> pair)
{
   String x = pair.getX();
   String y = pair.getY();
   System.out.println(x + ", " + y);
}

public static OrderedPair<Double> midPoint
       (OrderedPair<Double> p1, OrderedPair<Double> p2)
{
   double x = (p1.getX() + p2.getX())/2;
   double y = (p1.getY() + p2.getY())/2;
   return new OrderedPair<Double>(x, y);
}
```

Parameterized type interfaces

You can define an ordinary class type to implement a parameterized interface type. This example uses `Comparable<T>` from the Java API.

```
public class SaleItem implements Comparable<SaleItem> {
   private String description;
   private int barCode;

   public SaleItem (String description, int barCode) {
      this.description = description;
      this.barCode = barCode;
   }

   @Override
   public int compareTo(SaleItem x) {
      String thisData = description + barCode;
      String xData = x.description + x.barCode;
      return thisData.compareTo(xData);
   }
}
```

Wildcards

If you want a generic type argument that is independent of a particular parameterized, you can use a **type wildcard**, which is indicated by **<?>**. The following method prints all of the elements of its actual argument in reverse order.

```
public public void printReverse(OrderedPair<?> p) {
    Object x = p.getX();
    Object y = p.getY();
    System.out.println("Reverse: (" + y + ", " + x +")");
}
```

Since the compiler cannot know what parameterized type will be used for each call, the only type that you can use to declare the local variables is `Object`.

Restrictions on Generics

Some things you cannot do with parameterized/generic types:

- Cannot Cannot create a parameterized type with a primitive.
- Generic type definition: cannot declare a *static* variable with the type parameter.
- Cannot create arrays of parameterized types.

```
// OrderedPair<int> badPoint;
   OrderedPair<Integer> goodPoint;

// OrderedPair<Integer>[] badArray
//                        = new OrderedPair<Integer>[30];

   Object[] goodArray = new Object[30];
   goodArray[0] = new OrderedPair<String>("hi","bye");
```

Wrapper classes for primitives

You cannot declare an ArrayList with a primitive type:

```
ArrayList<int> bad = new ArrayList<int>(); // Won’t compile
```

Solution -- use a **wrapper class**:

```
ArrayList<Integer> measures = new ArrayList<Integer>();
```

Java has wrapper classes for all primitive numeric types:

- `Integer – int`
- `Double – double`
- `Float – float`
- `Long – long`
- `Byte, Short`

The Java compiler automatically converts primitives to their corresponding wrapper class types as needed. This conversion is called **boxing** (automatic = **auto boxing**).

Boxing examples

```
public static void printList(ArrayList<?> s) {
   for (Object o: s)
     System.out.print(o + “, “);
}

// Calling code
ArrayList<Integer> measures = new ArrayList<>();
measures.add(12);
measures.add(7);
measures.add(30);

int sum = measures.get(0) + measures.get(1); // sum is 19

if (measures.contains(7)) {
   // You can also add without boxing
   measures.add(2, new Integer(22));
   printList(measures);
}
```

Output: `12, 7, 22, 30,`