Object-Oriented Programming

Generics

Allowing operations not be to tied to a specific data type

Outline

- There are programming solutions that are applicable to a wide range of different data types
  - The code is exactly the same other than the data type declarations

- In Java, you can make use of **generic programming**:
  - A mechanism to specify solution without tying it down to a specific data type

- Let's define a class to:
  - Store a pair of integers, e.g. (**`74, -123`**)
  - Many usages, can represent 2D coordinates, range (min to max), height and weight, etc.

IntPair.java

```
1   class IntPair {
2
3     private int first;
4     private int second;
5
6     public IntPair(int first, int second) {
7       this.first = first;
8       this.second = second;
9     }
10
11    public int getFirst() {
12      return first;
13    }
14
15    public int getSecond() {
16      return second;
17    }
18  }
```

TestIntPair.java

```java
// This program uses the IntPair class to create an object
// containing the lower and upper limits of a range.
// We then use it to check that the input data fall within that range.
import java.util.Scanner;

public class TestIntPair {

  public static void main(String[] args) {
    IntPair range = new IntPair(-5, 20);
    Scanner sc = new Scanner(System.in);
    int input;

    do {
      System.out.printf("Enter a number in (%d to %d): ", range.getFirst(), range.getSecond());
      input = sc.nextInt();
    } while ((input < range.getFirst()) || (input > range.getSecond()));
  }
}
```

```
Enter a number in (-5 to 20): -10
Enter a number in (-5 to 20): 21
Enter a number in (-5 to 20): 12
```

- The **`IntPair`** class idea can be easily extended to other data types:
  - **`double`**, **`String`**, etc.

- The resultant code would be almost the same!

StringPair.java

```
1  class StringPair {
2
3    private String first;
4    private String second;
5
6    public StringPair(String first, String second) {
7      this.first = first;
8      this.second = second;
9    }
10
11   public String getFirst() {
12     return first;
13   }
14
15   public String getSecond() {
16     return second;
17   }
18 }
```

Only differences are the data type declarations

Pair.java

```
1   class Pair<T> {
2
3     private T first;
4     private T second;
5
6     public Pair(T first, T second) {
7       this.first = first;
8       this.second = second;
9     }
10
11    public T getFirst() {
12      return first;
13    }
14
15    public T getSecond() {
16      return second;
17    }
18  }
```

- Important restriction:
  - The generic type can be substituted by reference data type only
  - Hence, primitive data types are NOT allowed
  - Need to use wrapper class for primitive data type

TestGenericPair.java

```
1   public class TestGenericPair {
2     public static void main(String[] args) {
3       Pair<Integer> intPair = new Pair<Integer>(-5, 20);
4       Pair<String> stringPair = new Pair<String>("Turing", "Alan");
5
6       // You can have pair of any reference data types!
7       // Print out the integer pair
8       System.out.println("Integer pair: (" + intPair.getFirst()
9                          + ", " + intPair.getSecond() + ")";
10      // Print out the String pair
11      System.out.println("String pair: (" + stringPair.getFirst()
12                         + ", " + stringPair.getSecond() + ")";
13    }
14  }
```

- The formal generic type **<T>** is substituted with the actual data type supplied by the user:
  - The effect is similar to generating a new version of the **Pair** class, where **T** is substituted

- The following statement invokes auto-boxing

```
Pair<Integer> twoInt = new Pair<Integer>(-5, 20);
```

- **Integer** objects are expected for the constructor, but -5 and 20, of primitive type **int**, are accepted.
- ***Auto-boxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes
  - The primitive values -5 and 20 are converted to objects of **Integer**
- The Java compiler applies autoboxing when a primitive value is:
  - Passed as a parameter to a method that expects an object of the corresponding wrapper class
  - Assigned to a variable of the correspond wrapper class

- Converting an object of a wrapper type (e.g.: **Integer**) to its corresponding primitive (e.g: **int**) value is called auto-unboxing.
- The Java compiler applies auto-unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type
  - Assigned to a variable of the corresponding primitive type

```
int i = Integer.valueOf(5);     // auto-unboxing
Integer intObj = 7;             // auto-boxing
System.out.println("i = " + i);
System.out.println("intObj = " + intObj);
```

```
i = 5
intObj = 7
```

```
int a = 10;
Integer b = 10;                 // auto-boxing
System.out.println(a == b);
```

```
true
```

- We can have more than one generic type in a generic class
- Let's modify the generic pair class such that:
  - Each pair can have two values of **different data types**

NewPair.java

```
1  class NewPair<S, T> {
2
3    private S first;
4    private T second;
5
6    public NewPair(S first, T second) {
7      this.first = first;
8      this.second = second;
9    }
10
11   public S getFirst() {
12     return first;
13   }
14
15   public T getSecond() {
16     return second;
17   }
18 }
```

TestNewGenericPair.java

```
1  public class TestNewGenericPair {
2
3    public static void main(String[] args) {
4
5      NewPair<String, Integer> someone = new NewPair<String, Integer>("James Gosling", 55);
6
7      System.out.println("Name: " + someone.getFirst());
8      System.out.println("Age: " + someone.getSecond());
9    }
10 }
```

```
Name: James Gosling
Age: 55
```

- This NewPair class is now very flexible!
  - Can be used in many ways

- Generics was added in Java 5 to provide **compile-time type checking** and removing risk of ClassCastException that was common while working with collection classes.
- The whole collections framework was re-written to use generics for type-safety.
- The code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer.

GenericsDemo.java

```
1   List list = new ArrayList();
2   list.add("abc");
3   list.add(new Integer(5));  // OK
4
5   for (Object obj : list) {
6     // type casting leading to
7     // ClassCastException at runtime
8     String str = (String)obj;
9   }
```

- After Java 5, we use collection classes.
- Notice that at the time of list creation, we have specified that the type of elements in the list will be String. So if we try to add any other type of object in the list, the program will throw compile-time error.
- Also notice that in for loop, we don't need typecasting of the element in the list, hence removing the ClassCastException at runtime.

GenericList.java

```
1  List<String> listString = new ArrayList<String>();
2  // java 7? List<String> listString = new ArrayList<>();
3
4  listString.add("abc");
5  // listString.add(new Integer(5)); // compiler error
6
7  for (String str : listString) {
8    // no type casting needed, avoids ClassCastException
9  }
```

- Let's say we have a simple class GenericTypeOld
- Notice that while using this class, we have to use type casting and it can produce ClassCastException at runtime.

GenericTypeOld.java

```
1  public class GenericTypeOld {
2    private Object t;
3
4    public Object get() {
5      return t;
6    }
7
8    public void set(Object t) {
9      this.t = t;
10   }
11
12   public static void main(String args[]) {
13     GenericTypeOld type = new GenericTypeOld();
14     type.set("Pankaj");
15     String str = (String)type.get();
16     // type casting, error prone and can cause
17     // ClassCastException
18   }
19 }
```

- We will use java generic class to rewrite the same class.
- Notice the use of GenericsType class in the main method. We don't need to do type-casting and we can remove ClassCastException at runtime.
- When we don't provide the type, the type becomes Object and hence it's allowing both String and Integer objects. But, we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

GenericsType.java

```
1  public class GenericsType<T> {
2    private T t;
3
4    public T get() {
5      return this.t;
6    }
7
8    public void set(T t1) {
9      this.t = t1;
10   }
11
12   public static void main(String args[]) {
13     GenericsType<String> type = new GenericsType<>();
14     type.set("Pankaj");  // valid
15
16     GenericsType type1 = new GenericsType();  // raw type
17     type1.set("Pankaj");  // valid
18     type1.set(10);        // valid and auto-boxing support
19   }
20 }
```

- Comparable interface is a great example of Generics in interfaces. In similar way, we can create generic interfaces in java.

- We can also have multiple type parameters as in Map interface.

- Again we can provide parameterized value to a parameterized type also, for example

```
new HashMap<String, List<String>>();
```

is valid.

GenericsDemo.java

```
1  package java.lang;
2
3  import java.util.*;
4
5  public interface Comparable<T> {
6    public int compareTo(T o);
7  }
```

- Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language. So generics also comes with its own naming conventions.

- Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables.

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)
- S, U, V etc. – 2nd, 3rd, 4th types

- Sometimes we don't want the whole class to be parameterized, in that case, we can create java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too.

- Notice the *isEqual* method signature showing syntax to use generics type in methods.

- To use these methods in our java program, we can specify type while calling these methods or we can invoke them like a normal method. Java compiler is smart enough to determine the type of variable to be used, this facility is called **type inference**.

GenericsMethods.java

```java
public class GenericsMethods {
  // Java Generic Method
  public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2) {
    return g1.get().equals(g2.get());
  }

  public static void main(String args[]) {
    GenericsType<String> g1 = new GenericsType<>();
    g1.set("Pankaj");

    GenericsType<String> g2 = new GenericsType<>();
    g2.set("Pankaj");

    boolean isEqual = GenericsMethods.<String>isEqual(g1, g2);
    // above statement can be written simply as
    isEqual = GenericsMethods.isEqual(g1, g2);
    // This feature, known as type inference, allows you to invoke a generic method
    // as an ordinary method, without specifying a type between angle brackets.
    // Compiler will infer the type that is needed
  }
}
```

- Type parameters can be bounded. Bounded means "*restricted*", we can restrict types that can be accepted by a method.

- For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type all its superclasses (lower bound).

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
   ...
}
```

- The keyword *extends* is used here to mean that the type *T* extends the upper bound in case of a class or implements an upper bound in case of an interface.

- A type can also have multiple upper bounds as follows:

```
<T extends Number & Comparable>
```

- If one of the types that are extended by *T* is a class (i.e *Number*), it must be put first in the list of bounds. Otherwise, it will cause a compile-time error.

- We know that Java inheritance allows us to assign a variable A to another variable B if A is subclass of B. So we might think that any generic type of A can be assigned to generic type of B, but it's not the case.

- We are not allowed to assign MyClass<String> variable to MyClass<Object> variable because they are not related, in fact MyClass<T> parent is Object.

GenericsInheritance.java

```
1   public class GenericsInheritance {
2
3     public static void main(String[] args) {
4       String str = "abc";
5       Object obj = new Object();
6       obj = str;  // works because String is-a Object
7
8       MyClass<String> myClass1 = new MyClass<String>();
9       MyClass<Object> myClass2 = new MyClass<Object>();
10      // myClass2 = myClass1; // compilation error since
11      // MyClass<String> is not a MyClass<Object>
12      obj = myClass1; // MyClass<T> parent is Object
13    }
14
15    public static class MyClass<T> {
16    }
17  }
```

- We can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.
- For example, ArrayList<E> implements List<E> that extends Collection<E>, so ArrayList<String> is a subtype of List<String> and List<String> is subtype of Collection<String>.
- The subtyping relationship is preserved as long as we don't change the type argument.

```
interface MyList<E, T> extends List<E> {

}
```

- The subtypes of List<String> can be MyList<String, Object>, MyList<String, Integer> and so on.

- Question mark (?) is the wildcard in generics and represent an unknown type.
- The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type.
- We can’t use wildcards while invoking a generic method or instantiating a generic class.
- In the following sections, we will learn about upper bounded wildcards, lower bounded wildcards, and wildcard capture.

- Upper bounded wildcards are used to relax the restriction on the type of variable in a method. Suppose we want to write a method that will return the sum of numbers in the list, so our implementation will be something like this.

```
public static double sum(List<Number> list) {
  double sum = 0;
  for (Number n : list) {
    sum += n.doubleValue();
  }
  return sum;
}
```

- Now the problem with above implementation is that it won't work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when an upper bounded wildcard is helpful. We use generics wildcard with **extends** keyword and the **upper bound** class or interface that will allow us to pass argument of upper bound or it's subclasses types.

- The above implementation can be modified to work with List of Integers or Doubles.
- It's similar like writing our code in terms of interface, we can use all the methods of upper bound class Number.
- Note that with upper bounded list, we are not allowed to add any object to the list except *null*. If we will try to add an element to the list inside the sum method, the program won't compile.

GenericsWildcards.java

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class GenericsWildcards {
5    public static void main(String[] args) {
6      List<Integer> ints = new ArrayList<>();
7      ints.add(3);
8      ints.add(5);
9      ints.add(10);
10     double sum = sum(ints);
11     System.out.println("Sum of ints = " + sum);
12   }
13
14   public static double sum(List<? extends Number> list) {
15     double sum = 0;
16     for (Number n : list) {
17       sum += n.doubleValue();
18     }
19     return sum;
20   }
21 }
```

- Sometimes we have a situation where we want our generic method to be working with all types, in this case, an unbounded wildcard can be used. Its same as using <? extends Object>.

```
public static void printData(List<?> list) {
  for (Object obj : list) {
    System.out.print(obj + "::");
  }
}
```

- We can provide List<String> or List<Integer> or any other type of Object list argument to the *printData* method.

- Similar to upper bound list, we are not allowed to add anything to the list.

- Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as List<Integer> but it will be tied up with Integers whereas List<Number> and List<Object> can also hold integers, so we can use a lower bound wildcard to achieve this. We use generics wildcard (?) with **super** keyword and lower bound class to achieve this.

- We can pass lower bound or any supertype of lower bound as an argument, in this case, java compiler allows to add lower bound object types to the list.

```
public static void addIntegers(List<? super Integer> list) {
  list.add(new Integer(50));
}
```

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;
// OK. List<? extends Integer> is a subtype of List<? extends Number>
```

- Generics were added to Java to ensure type safety and to ensure that generics wouldn't cause overhead at runtime, the compiler applies a process called *type erasure* on generics at compile time.
- Type erasure removes all type parameters and replaces it with their bounds or with *Object* if the type parameter is unbounded. Thus the bytecode after compilation contains only normal classes, interfaces and methods thus ensuring that no new types are produced. Proper casting is applied as well to the *Object* type at compile time.
- This is an example of type erasure:

```
public <T> List<T> genericMethod(List<T> list) {
  return list.stream().collect(Collectors.toList());
}
```

- With type erasure, the unbounded type *T* is replaced with *Object* as follows:

```
// for illustration
public List<Object> withErasure(List<Object> list) {
  return list.stream().collect(Collectors.toList());
}
```

- If the type is bounded, then the type will be replaced by the bound at compile time:

```
Public <T extends Building> void genericMethod(T t) {
   ...
}
```

- After compilation:

```
public void genericMethod(Building t) {
   ...
}
```

- **A restriction of generics in Java is that the type parameter cannot be a primitive type.**

- For example, the following doesn't compile:

```
List<int> list = new ArrayList<>();
list.add(17);
```

- To understand why primitive data types don't work, let's remember that **generics are a compile-time feature**, meaning the type parameter is erased and all generic types are implemented as type *Object*.
- As an example, let's look at the *add* method of a list:

```
List<Integer> list = new ArrayList<>();
list.add(17);
```

- The signature of the *add* method is:

```
boolean add(E e);
```

- And will be compiled to:

```
boolean add(Object e);
```

- Therefore, type parameters must be convertible to *Object*. **Since primitive types don't extend *Object*, we can't use them as type parameters.**

- However, Java provides boxed types for primitives, along with auto-boxing and auto-unboxing to unwrap them:

```
Integer a = 17;
int b = a;
```

- So, if we want to create a list which can hold integers, we can use the wrapper:

```
List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

- The compiled code will be the equivalent of:

```
List list = new ArrayList<>();
list.add(Integer.valueOf(17));
int first = ((Integer)list.get(0)).intValue();
```

- Generics are useful when the code remains unchanged other than differences in data types

- When you declare a generic class/method, make sure that the code is valid for all possible data types

Using the Vector and ArrayList classes

Drawback of Arrays

- Array, as discussed in previous weeks, has a major drawback:
  - Once initialized, the array size is fixed
  - Reconstruction is required if the array size changes
  - To overcome such limitation, we can use some classes related to array
- Java has an **Array** class
  - Check API documentation and explore it yourself
- However, we will not be using this **Array** class much; we will be using other classes such as **Vector** and **ArrayList**
  - Both provide re-sizable array, i.e. array that is growable
  - Both are implementations of the List interface

Class for dynamic-size arrays

- Java offers a Vector class to provide:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods

- Use array if the size is fixed; use Vector if the size may change.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	add(int index, E element)	Inserts the specified element at the specified position in this Vector.
boolean	add(E e)	Appends the specified element to the end of this Vector.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified Collection into this Vector at the specified position.
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that tl
void	addElement(E obj)	Adds the specified component to the end of this vector, increasing its size by one.
int	capacity()	Returns the current capacity of this vector.
void	clear()	Removes all of the elements from this Vector.
Object	clone()	Returns a clone of this vector.
boolean	contains(Object o)	Returns true if this vector contains the specified element.
boolean	containsAll(Collection<?> c)	Returns true if this Vector contains all of the elements in the specified Collection.
void	copyInto(Object[] anArray)	Copies the components of this vector into the specified array.
E	elementAt(int index)	Returns the component at the specified index.
Enumeration<E>	elements()	Returns an enumeration of the components of this vector.

PACKAGE

```
import java.util.Vector;
```

SYNTAX

```
// Declaration of a Vector reference
Vector<E> myVector;

// Initialize a empty Vector object
myVector = new Vector<E>();
```

Commonly Used Method Summary

boolean	*isEmpty*() Tests if this vector has no components.
int	*size*() Returns the number of components in this vector.

Commonly Used Method Summary (continued)

boolean	***add***(**E o**) Appends the specified element to the end of this Vector.
void	***add***(**int index, E element**) Inserts the specified element at the specified position in this Vector.
E	***remove***(**int index**) Removes the element at the specified position in this Vector.
boolean	***remove***(**Object o**) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
E	***get***(**int index**) Returns the element at the specified position in this Vector.
int	***indexOf***(**Object elem**) Searches for the first occurrence of the given argument, testing for equality using the equals method.
boolean	***contains***(**Object elem**) Tests if the specified object is a component in this vector.

Example

TestVector.java

```
1  import java.util.Vector;
2
3  public class TestVector {
4    public static void main(String[] args) {
5      Vector<String> courses = new Vector<String>();
6      courses.add("CS1020");
7      courses.add(0, "CS1010");
8      courses.add("CS2010");
9      System.out.println(courses);
10     System.out.println("At index 0: " + courses.get(0));
11     if (courses.contains("CS1020")) {
12       System.out.println("CS1020 is in courses");
13     }
14     courses.remove("CS1020");
15     for (String c : courses) {
16       System.out.println(c);
17     }
18   }
19 }
```

Vector class has a nice **toString()** method that prints all elements

The enhanced for-loop is applicable to Vector objects too!

```
Output:
[CS1010, CS1020, CS2010]
At index 0: CS1010
CS1020 is in courses
CS1010
CS2010
```

Another class for dynamic-size arrays

- Java offers an ArrayList class to provide similar features as Vector:
  - Dynamic size
    - expands or shrinks automatically
  - Generic
    - allows any reference data types
  - Useful predefined methods
- Similarities:
  - Both are index-based and use an array internally
  - Both maintain insertion order of element
- So, what are the differences between Vector and ArrayList?
  - This is one of the most frequently asked questions, and at interviews!

- Differences between Vector and ArrayList

Vector	ArrayList
Since JDK 1.0	Since JDK 1.2
Synchronised * (thread-safe)	Not synchronised
Slower (price of synchronisation)	Faster (≈20 – 30%)
Expansion: default to double the size of its array (can be set)	Expansion: increases its size by ≈50%

- ArrayList is preferred if you do not need synchronisation
  - Java supports multiple threads, and these threads may read from/write to the same variables, objects and resources. Synchronisation is a mechanism to ensure that Java thread can execute an object's synchronised methods one at a time.
- When using Vector /ArrayList, always try to initialise to the largest capacity that your program will need, since expanding the array is costly.
  - Array expansion: allocate a larger array and copy contents of old array to the new one

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
boolean	add(E e)	Appends the specified element to the end of this list.
boolean	addAll(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specifie
boolean	addAll(Collection<? extends E> c)	Appends all of the elements in the specified collection to the end of this list, in the order
void	clear()	Removes all of the elements from this list.
Object	clone()	Returns a shallow copy of this ArrayList instance.
boolean	contains(Object o)	Returns true if this list contains the specified element.
void	ensureCapacity(int minCapacity)	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hol
boolean	equals(Object o)	Compares the specified object with this list for equality.
void	forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been
E	get(int index)	Returns the element at the specified position in this list.
int	hashCode()	Returns the hash code value for this list.

PACKAGE

```
import java.util.ArrayList;
```

SYNTAX

```
// Declaration of a ArrayList reference
ArrayList<E> myArrayList;

// Initialize a empty ArrayList object
myArrayList = new ArrayList<E>();
```

Commonly Used Method Summary

boolean	*isEmpty*() Returns **true** if this list contains no element.
int	*size*() Returns the number of elements in this list.

Commonly Used Method Summary (continued)	
boolean	***add***(**E e**) Appends the specified element to the end of this list.
void	***add***(**int index, E element**) Inserts the specified element at the specified position in this list.
E	***remove***(**int index**) Removes the element at the specified position in this list.
boolean	***remove***(**Object o**) Removes the first occurrence of the specified element from this list, if it is present.
E	***get***(**int index**) Returns the element at the specified position in this list.
int	***indexOf***(**Object o**) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	***contains***(**Object elem**) Returns **true** if this list contains the specified element.

TestArrayList.java

```java
import java.util.ArrayList;
import java.util.Scanner;

public class TestArrayList {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    ArrayList<Integer> list = new ArrayList<Integer>();
    System.out.println("Enter a list of integers, press ctrl-d to end.");
    while (sc.hasNext()) {
      list.add(sc.nextInt());
    }
    System.out.println(list); // using ArrayList's toString()

    // Move first value to last
    list.add(list.remove(0));
    System.out.println(list);
  }
}
```

```
Output:
Enter a list ... to end.
31
17
-5
26
50
(user pressed ctrl-d here)
[31, 17, -5, 26, 50]
[17, -5, 26, 50, 31]
```

Thank you!