Generics

Java and type checking

- ✔ Java is **typesafe**: attempts to use an object in a way inconsistent with its type are detected as errors

- ✔ These errors can be detected at *compile time*, or at *run time*
- ✔ It is better to detect them at compile time!
  - ✗ The compiler detects and localizes errors automatically, without having to run and test your code
  - ✗ Run time errors require thorough testing to detect and you may not catch all of them before your code ships

- ✔ The main advantage of Java generics is that they permit more errors to be detected at compile-time
  - ✗ Though not all of them! Some can still occur at run time

- ✔ But in any case, Java with generics is still typsafe: attempts to use an object in a way inconsistent with its type are detected as errors, one way or another

Using Java generics

- In Java 5, a “generic” data structure can be created specifying the type of object it is to hold.
- Attempts to store another type of object in the structure, or to use an object taken from the structure as another type, will generate a compile-time error or warning
- For example, you can create a Vector and specify that it can hold only Integers:

```
Vector<Integer> v = new Vector<Integer>();

v.add(3);
v.add(10);
v.add(“Hello”);                    // compile time error
```

- Now taking an object out of the Vector does not need a cast:

```
Integer myInt = v.get(0);
```

- And in fact automatic boxing and unboxing of primitive types is now done:

```
v.add(44);
int i = v.get(0);
```

ArrayList <String> list = new ArrayList<String>();

- The <String> symbol used twice in the above line means that the elements of the collection must be String type.
- The type ArrayList <String> are called *parameterized types* or *generic types*.
- Specifying the element type this way causes the compiler to prevent any non-String objects from being inserted into the collection, making it *type-safe*.
- The JCF uses type parameters in most of its classes and interfaces.

Using type parameters with ArrayList

THIS code:

```
ArrayList<String> thisList = new ArrayList<String>
```

Means ArrayList:

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
      // more code
}
```

Is treated by the compiler as:

```
public class ArrayList<String> extends AbstractList<String>... {

      public boolean add(String o)
      // more code
}
```

Linked List class

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `List`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`.

Generics

- *generics*, which are particularly appropriate for implementing collections
- It allows to hold any type of object
- Enable you to write a placeholder instead of an actual class type
- The generic type placeholder is specified in angle brackets in the class header and an Interface:

```
class Box<T>
{
    // declarations and code that refer to T
}
```

- Any identifier can be used, but T (for Type) or E (for element) have become standard practice

Example using type argument

```
public interface Pairable <T>
{
public T getFirst();
public T getSecond();
public void changeOrder();
}
```

```
public class OrderedPair<T> implements Pairable <T> {
private T first, second;
public OrderedPair(T firstItem, T secondItem)
{
first = firstItem;
second = secondItem;
}
public T getFirst()
{
return first;
}
public T getSecond()
{
return second;
}
public void changeOrder()
{
T temp = first;
first = second;
second = temp;
}
public String toString()
{
return "(" + first + "," + second + ")";
}
}
```

```
public class PairDemo {
public static void main(String args[])
{
OrderedPair obj = new OrderedPair("Hello","World");
System.out.println("First = " + obj.getFirst());
System.out.println("Second = " + obj.getSecond());
obj.changeOrder();
System.out.println("Change Order = " + obj);
System.out.println("First = " + obj.getFirst());
System.out.println("Second = " + obj.getSecond());

OrderedPair obj1 = new OrderedPair(10,"Welcome");
System.out.println("First = " + obj1.getFirst());
System.out.println("Second = " + obj1.getSecond());
obj1.changeOrder();
System.out.println("Change Order = " + obj1);
System.out.println("First = " + obj1.getFirst());
System.out.println("Second = " + obj1.getSecond());
}
}
```

Output

Change Order = (World,Hello)

First = World

Second = Hello

First = 10

Second = Welcome

Change Order = (Welcome,10)

First = Welcome

Second = 10

Generic Methods

- In addition to generic types, type parameters can also be used to define *generic methods*, identified by the generic parameter specifier <T> placed in front of the return type.
- The method is identified as generic by the <E> specifier allows the type parameter E to be used in place of an actual type in the method block.

```
public class GenericMethod {
        public static void main(String[] args) {
         args = new String[]{"CA", "US", "MX", "HN", "GT"};
                print(args);
                Integer[] x = new Integer[]{10,20,30,40,50};
                print(x);
                }
static <E> void print(E[] a) {
                for (E ae : a) {
                System.out.printf("%s ", ae);
                }
                System.out.println();
                }
```

GENERIC WILDCARDS

- The symbol ? can be used as a wildcard, in place of a generic variable. It stands for “unknown type,” and is called the *wildcard type*.
- Example :

```
static void display(Collection<?> c) {
                    for (Object o : c) {
                    System.out.printf("%s ", o);
                    }
                    System.out.println();
                    }
```

The ? With Bounded Wildcard

- an *upper bounded* wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its *upper bound :* <? **extends** superclass>
- In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its *lower bound*: <? **super** subclass>.