Introduction to Java for C++ Programmers

Professor: Mahboob Ali

What is Generics?

- Generics adds stability to your code by detecting bugs on the compile time.
- Generics is the capability to parameterize types.
- Generics enable types to be parameter when defining classes, interfaces and methods.
- We are all familiar with passing arguments in methods, where values travel in those arguments, now with the help of generics we can <u>Pass types</u> as arguments.

Motivation

Polymorphism promotes

```
class Store{
    private Bookmark a;

    public void set(Bookmark a){
        this.a = a;
    }

    public Bookmark get(){
        return a;
    }
}
```

Can only hold **Bookmark** objects or its subtype objects

✓ **Type is hardcoded**

More Generalized Form

```
class Store{
    private Object a;

    public void set(Object a){
        this.a = a;
    }

    public Object get(){
        return a;
    }
}
```

John:
Store store = new Store();
store.*set*(**new Date()**); // java.util.Date
…
Date date = **(Date)** store.*get*() //Cast

Bob:
store.*set*(**new Date()**); // java.sql.Date

John:
Date date = **(Date)** store.*get*();
//java.util.Date

ClassCastException

✓Too generic **✓Explicit Casting** **✓Runtime Exception**

Generics was introduced to solve this

Generics is purely compile time concept

```java
class Store <T>{
    private T a;

    public void set(T a){
        this.a = a;
    }

    public T get(){
        return a;
    }
}
```

John:

```java
Store<Date> store = new Store<Date> (); //java.util
store.set(new Date());
…
Date date = store.get() //no Casting
```

Bob:

```java
store.set(new Date()); // java.sql.Date
        //compiler error
```

Mike:

```java
Store<Book> store = new Store<Book> ();
store.set(new Book());
```

✓ **Type safety at compile-time**

✓ **cleaner code**

✓ **Expressive code**

✓ **Generics**

Generics and Parameterized Types

Generic Type

Class or Interface with ***type parameters***

class ClassName**<T1, T2, T3, …>** {… …}

✓ Type of *instance variables*
✓ Type of *parameters, local variables, return types*

```
class Store <T>{
    private T a;

    public void set(T a){
        this.a = a;
    }

    public T get(){
        return a;
    }
}
```

Type parameter

Instance variable

Method parameter

Return type

Generic Type and Parameterized Type

```
class Store<T>{ …. }
```

Generic type

Type parameter (or Formal Type parameter

Type argument (or Actual Type parameter

Parameterized type

```
Store<String> stringStore = new Store<String>()
Store<Date> dateStore = new Store<Date>()
Store<List<Date> > dateListStore = new Store<List<Date>>()
Store<Book> bookStore = new Store<>()
```

Instantiating the generic type

By replacing type **T** with some concrete value such String or Data or List<Date> or Book, we are invoking the generic.

Diamond notation (Java 7)

Type Parameter Naming Conventions

Use **single, uppercase** letters

E – Element (Collections)
K – Key, **V** – Value (Maps)
N – Numbers
T – Type (usually in non-collection)
S, U, V – 2nd, 3rd, 4th types

Subtyping of Generic Types

```
interface Container<T> {
    void set(T a);
    T get();
}
```

```
class Store<T> implements Container<T> {
    private T a;

    public void set(T a) {
        this.a = a;
    }

    public T get() {
        return a;
    }
}
```

Parameterized type

Container<String> store = new Store<>();

Multiple Type Parameters

- You can also have multiple Type Parameters as well. Example

```
public interface Pair<K, V>{

        public K getKey();

        public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V>{

        private K key;

        private V value;

        public OrderedPair(K key, V value){

                this.key = key;

                this.value = value;

        }

        public K getKey() {return key;}

        public V getValue() {return value;}     }
```

- Instasiation of the OrderedPair class

```
OrderedPair<String, Integer> p1
= new OrderedPair<>("One", 1);


OrderedPair<String, String> p2
= new OrderedPair<>("Hello",
"world");
```

Why Generics: Benefits ??

Stronger Type checking :

- Fixing Error at run-time or Fixing error at compile time?

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32); //Compile Time Error
```

Casting can be eliminated:

- No more object types casting.

<u>with type cast</u>

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

<u>without type case</u>

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

Type Safety:

- Holds only single type of objects, doesn’t allow to store other objects.

Non Generic Example:

```
import java.util.*;
public class ArrayListWithoutGenericsTest {
   public static void main(String[] args) {
      List strLst = new ArrayList();
      strLst.add("alpha"); // String upcast to Object implicitly
      strLst.add("beta");
      strLst.add("charlie");
      Iterator iter = strLst.iterator();
      while (iter.hasNext()) {
    // need to explicitly downcast Object back to String
        String str = (String)iter.next();
        System.out.println(str);
      }
      // Compiler/runtime cannot detect this error
      strLst.add(new Integer(1234));
      // compile ok, but runtime ClassCastException
      String str = (String)strLst.get(3);
   }
}
```

We could use an **instanceof** operator to check for proper type before down-casting. Again checking will be done on run-time

Generic Class (Another Example)

GenericBox.java

```
public class GenericBox<E> {
    private E content; // Private variable
    public GenericBox(E content) {// Constructor
            this.content = content;
    }
    public E getContent() {
            return content;
    }
    public void setContent(E content) {
            this.content = content;
    }
    public String toString() {
            return content + " (" + content.getClass() + ")";
  } }
```

toString() reveals the actual type of the contents

```java
public class TestGenericBox {
   public static void main(String[] args) {

    GenericBox<String> box1 = new GenericBox<>("Hello");
    // no explicit downcasting needed
    String str = box1.getContent();
    System.out.println(box1);

    // autobox int to Integer
    GenericBox<Integer> box2 = new GenericBox<>(123);
    // downcast to Integer, autoboxing to int
    int i = box2.getContent();
    System.out.println(box2);

   // autobox double to Double
   GenericBox<Double> box3 = new GenericBox<>(55.66);
   // downcast to Double, autoboxing to double
   double d = box3.getContent();
   System.out.println(box3);
  }
}
```

```
Hello (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)
```

Type Erasure

- When a generic type is instantiated, the compiler translates those types by a technique called *type erasure*

- In previous example the compiler replaces all reference to parameterized type E with Object, performs the type check, and insert the required downcast operators.

```
// A dynamically allocated array with generics
public class MyGenericArrayList<E> {
   private int size; // number of elements
   private Object[] elements;
   public MyGenericArrayList() { // constructor
      elements = new Object[10];
    }
   public void add(E e) {
      if (size < elements.length) {
            elements[size] = e;
      }
      else { // allocate a larger array and add the element…. }
      ++size;
   }
   public E get(int index) {
      if (index >= size) throw new IndexOutOfBoundsException("Index: "
                              + index + ", Size: " + size);
          return (E)elements[index];
   }
   public int size() {
         return size;
   }
}
```

```
public class MyGenericArrayListTest {
   public static void main(String[] args) {
     // type safe to hold a list of Strings
     MyGenericArrayList<String> strLst = new MyGenericArrayList<>();
     strLst.add("alpha"); // compiler checks if argument is of type String
     strLst.add("beta");

     for (int i = 0; i < strLst.size(); ++i) {
           // compiler inserts the downcasting operator (String)
           String str = strLst.get(i);
           System.out.println(str);
      }

    // compiler detected argument is NOT String, issues compilation error
      strLst.add(new Integer(1234));
   }
}
```

Raw Types

- *Raw Type* is the name of the a generic class or interface without any type arguments. For example

```
public class Store<T>{
      public void setStore(T t) { /* … */}
      // …
}
```

```
Store<String> strStore = new Store<>();
```
Creating a parameterized type store

```
Store rawStore = new Store();
```
Creating a ***raw type*** store

- We should avoid raw types. Why?
  - They are not type safe, you possibly get runtime error.
  - They require proper casting
  - They act as normal object instantiation

Generic method

- Type parameters can also be declared within method and constructor signatures to create *generic method*
- Type parameter's scope is limited to the method in which it is declared.

```
public class GenericsMethods {
  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("hello");
   GenericsType<String> g2 = new GenericsType<>();
   g2.set("hello");
   boolean isEqual = GenericsMethods.<String>isEqual(g1, g2);

   //above statement can be written simply as
   isEqual = GenericsMethods.isEqual(g1, g2);
   //without specifying a type between angle brackets.
  }
}
```

Bounded Type Parameter

- Sometimes we want to bound the parameterized type.
- For example a method that works on numbers only want to accept instance of Numbers or its subclasses.

<TypeParameter **extends** *bound1 & bound 2 & …* > *{…}*

class GenericDemo **<T extends List>**{

…

}

List the type parameter name followed by **extends**

```java
public class Box<T> {
      private T t;
      public void set(T t) { this.t = t; }
      public T get() { return t; }
      public <U extends Number> void inspect(U u){
            System.out.println("T: " + t.getClass().getName());
            System.out.println("U: " + u.getClass().getName());
      }
      public static void main(String[] args) {
            Box<Integer> integerBox = new Box<Integer>();
            integerBox.set(new Integer(10));
                                         // error: this is still String!
            integerBox.inspect("some text");
      }
}
```

Bounded type parameters allow you to invoke methods defined in the bounds

```
public class NumbersClass <T extands Integer{
   private T n;
   public NumbersClass(T n){
      this.n = n;
   }
   public boolean isEven(){
      return n.intValue() % 2 == 0;
   }
}
```

Here *isEven()* method invokes *intValue* method defined in the **Integer** Class via *n*.

Wildcards

- Consider the problem,

```
ArrayList<object> lst = new ArrayList<String>();
```

Obvious “incompatible types” error will occur

Look again…

This error is against our intuition on polymorphism, as we often assign a subclass instance to a superclass reference.

```
import java.util.*;
public class TestGenericWildcard {
     public static void printList(List<Object> lst) {
          // accept List of Objects only, not List of subclasses of object
    for (Object o : lst)
        System.out.println(o); }
    public static void main(String[] args) {
        List<Object> objLst = new ArrayList<Object>();
        objLst.add(new Integer(55));
        printList(objLst); // matches
        List<String> strLst = new ArrayList<String>();
        strLst.add("one");
        printList(strLst); // compilation error
    } }
```

Solution:

To resolve this problem, a **wildcard (?)** is provided in generics, stands for any unknown type.

```
public static void printList(List<?> lst) {
    for (Object o : lst)
        System.out.println(o);
}
```

Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles.
To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) {
        ...
    }
}
public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) {
        ...
    }
}
These classes can be drawn on a canvas:
public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }   }
```

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>.
What we really want is for the method to accept a list of **any** kind of shape:

```
public void drawAll(List<? extends Shape> shapes)
      { ... }
```

There is a small but very important difference here:
we have replaced the type List<Shape> with List<? **extends** Shape>. Now drawAll() will accept lists of any **<u>subclass</u>** of Shape, so we can now call it on a List<Circle> if we want.

Restriction ~ Primitives

Type argument cannot be a primitive

```
Store<int> intStore = new Store<int>();
```

Restriction ~ Static Context

Type argument cannot be used in static context

Type of static variables:

```
public  class Device<T>{
      private static T deviceType;
}

Device<Smartphone> phone = new Device<>();
Device<Pager> pager = new Device<>();
```

T? Smartphone or pager