Generics

- What are generics ?
- Generic Class – an Example
    - Need for Generics
    - Generic with Two parameters
    - Bounded Types
    - Wildcard Arguments
- Generic Methods
- Generic Constructors
- Inheritance using Generics

What are Generics ?

- In pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects.
- The problem was that they could not do so with *type safety*.

What are Generics ?

- The term generics means *parameterized types*.
- It is possible to create a single class, for example, that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called ***generic***, as in generic class or generic method.

Erasure

- The compiler removes all generic type information.
- Substituting the necessary casts, make your code *behave as if* a specific version of Generic were created.
- Thus, there is really only *one version of Generic* that actually exists in your program.
- The process of removing generic type information is called *erasure*.

Generics – Example

```
class Gen<T> {
   T ob;
   Gen(T o) {
      ob = o;
   }
   T getob() {
      return ob;
   }
   void showType() {
      System.out.println("Type of T is " +
                             ob.getClass().getName());
   }
}

class GenDemo {
   public static void main(String args[]) {
   Gen<Integer> iOb;    //CREATE Gen reference FOR Integer
```

type parameter

Must be a class type

Generics – Example

```
   iOb = new Gen<Integer>(88);
   iOb.showType();
   int v = iOb.getob();
   System.out.println("value: " + v);
   System.out.println();
   Gen<String> strOb = new Gen<String> ("Generics Test");
   strOb.showType();
   String str = strOb.getob();
   System.out.println("value: " + str);
   }
}
-----------------------------------------------------------
```

type safety

autoboxing

Output:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Generics – limitations

- Works only with **objects**
- It is possible to pass any class type to T, but you cannot pass a primitive type to a type parameter:

```
Gen<int> intOb = new Gen<int>(53); // Error
```

- Can use the type wrappers to encapsulate a primitive type.
- Java’s autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generics – limitations

- **Generic Types Differ Based on Their Type Arguments**
- A reference of *one specific version* of a generic type is not type compatible with *another version* of the **same** generic type.

```
    iOb = strOb; // Wrong!
```

- Generics add type safety and prevent errors.

Need for Generics

```
class NonGen {
   Object ob;
   NonGen(Object o) {
      ob = o;
   }

   Object getob() {
      return ob;
   }
   void showType() {
      System.out.println("Type of ob is "+
                             ob.getClass().getName());
   }
}

class NonGenDemo {
   public static void main(String args[]) {
      NonGen iOb;
      iOb = new NonGen(88);
```

Need for Generics

```
    iOb.showType();
    int v = (Integer) iOb.getob();
    System.out.println("value: " + v);
    System.out.println();

    NonGen strOb = new NonGen("Non-Generics Test");
    strOb.showType();
    String str = (String) strOb.getob();
    System.out.println("value: " + str);

    // This compiles, but is conceptually wrong!
    iOb = strOb;
    v = (Integer) iOb.getob(); // run-time error!
    }
}
```

iOb now refers to String not an Integer

type-mismatch error

Generics advantages

- Generics added the type safety
- No longer necessary to *explicitly* employ *casts* to translate between **Object** and the type of data that is actually being operated upon.
- All casts are automatic and implicit – autoboxing

Generics advantages

- Without the use of generics, the Java compiler has no way to know the type-mismatch !
- The ability to create type-safe code in which type-mismatch errors are caught at compile time – key advantage
- Through generics, run-time errors are converted into compile-time errors.

Generics with Two Parameters

```
class TwoGen<T, V> {
T ob1;
V ob2;
TwoGen(T o1, V o2) {
   ob1 = o1;
   ob2 = o2;
   }
void showTypes() {
   System.out.println("Type of T is " +
   ob1.getClass().getName());
   System.out.println("Type of V is " +
   ob2.getClass().getName());
   }
T getob1() {
   return ob1;
   }
V getob2() {
   return ob2;
   }
}
```

Generics with Two Parameters

```
class SimpGen {
   public static void main(String args[]) {
      TwoGen<Integer, String> tgObj =
               new TwoGen<Integer, String>(88, "Generics");
      tgObj.showTypes();

      int v = tgObj.getob1();
      System.out.println("value: " + v);
      String str = tgObj.getob2();
      System.out.println("value: " + str);
   }
}
-----------------------------------------------------------
Output:
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Bounded Types

- Create a generic class that contains a method that returns the *average of an array of numbers*.
- Use the class to obtain the average of an *array of any type of number*, including *integers*, *floats*, and *doubles*.

Bounded Types

```
class Stats<T> {
   T[] nums;

   Stats(T[] o) {
      nums = o;
   }

   double average() {
      double sum = 0.0;
      for(int i=0; i < nums.length; i++)
         sum += nums[i].doubleValue();
      return sum / nums.length;
    }
}
```

What is the ERROR !

Bounded Types

```
class Stats<T> {
   T[] nums;

   Stats(T[] o) {
      nums = o;
   }

   double average() {
      double sum = 0.0;
      for(int i=0; i < nums.length; i++)
         sum += nums[i].doubleValue();
      return sum / nums.length;
    }
}
```

error: cannot find symbol

Bounded Types

- Compiler – no way to know that Stats uses only numeric !
- But intended to pass only numeric types to T.
- Needed some way to *ensure* that *only* numeric types are actually passed.
- Solution: use of an *extends* clause when specifying the type parameter

```
        <T extends superclass>
```

- Where T can only be replaced by superclass, or subclasses of superclass.
- Superclass defines an inclusive, upper limit.

Bounded Types

```
class Stats<T extends Number> {
   T[] nums;
   Stats(T[] o) {
      nums = o;
   }
   double average() {
      double sum = 0.0;
      for(int i=0; i < nums.length; i++)
         sum += nums[i].doubleValue();
      return sum / nums.length;
    }
}

class BoundsDemo {
public static void main(String args[]) {
   Integer inums[] = { 1, 2, 3, 4, 5 };
   Stats<Integer> iob = new Stats<Integer>(inums);
   double v = iob.average();
   System.out.println("iob average is " + v);
```

Bounded Types

```
   Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
   Stats<Double> dob = new Stats<Double>(dnums);
   double w = dob.average();
   System.out.println("dob average is " + w);

// compile-time error, String is not subclass of Number
// String strs[] = { "1", "2", "3", "4", "5" };
// Stats<String> strob = new Stats<String>(strs);
   }
}
-----------------------------------------------------------
Output:

Average is 3.0
Average is 3.3
```

Wildcard Arguments

- To write a method to find the two objects contains arrays that yield the same average.
- No matter what type of data each object holds!
- What do you specify for `Stats`' *type parameter* when you declare a parameter of that type?

Wildcard Arguments

```
boolean sameAvg(Stats<T> ob)
{
   if(average() == ob.average())
      return true;
   return false;
}
----------------------------------------------------------
```

How to specify type parameter?

Will work only with other Stats objects whose *type is the same as the invoking object*.

If the invoking object is of type **Stats<Integer>**,

then the parameter ob must also be of type **Stats<Integer>**.

Can't be used to compare the average of an object of type **Stats<Double>** with **Stats<Short>** - for example.

Wildcard Arguments

```
boolean sameAvg(Stats<?> ob) {
   if(average() == ob.average())
      return true;
   return false;
}
----------------------------------------------------------
```

Wildcard argument

The wildcard argument is specified by the ? and it represents an unknown type.

Stats<?> matches any Stats object

Wildcard Arguments

```
class Stats<T extends Number> {
T[] nums;      // array of Number or subclass
Stats(T[] o) {            // a reference to an array
   nums = o;              // of type Number or subclass.
   }

double average() {
   double sum = 0.0;
   for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();
   return sum / nums.length;
   }
// Determine if two averages are the same.
boolean sameAvg(Stats<?> ob) {
   if(average() == ob.average())
      return true;
   return false;
   }
}
```

Wildcard Arguments

```
class WildcardDemo {
public static void main(String args[]) {
   Integer inums[] = { 1, 2, 3, 4, 5 };
   Stats<Integer> iob = new Stats<Integer>(inums);
   double v = iob.average();
   System.out.println("iob average is " + v);

   Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
   Stats<Double> dob = new Stats<Double>(dnums);
   double w = dob.average();
   System.out.println("dob average is " + w);

   Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
   Stats<Float> fob = new Stats<Float>(fnums);
   double x = fob.average();
   System.out.println("fob average is " + x);
```

Wildcard Arguments

```
// See which arrays have same average.
System.out.print("Averages of iob and dob ");
if(iob.sameAvg(dob))
   System.out.println("are the same.");
else
   System.out.println("differ.");

System.out.print("Averages of iob and fob ");
if(iob.sameAvg(fob))
   System.out.println("are the same.");
else
   System.out.println("differ.");
   }
}
-------------------------------------------------------
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

Generic Method

- It is possible to create a generic method that is enclosed within a non-generic class.
- Syntax:

```
<type-param-list> ret-type meth-name (param-list)
```

- The type parameters are declared before the return type of the method.
- Note that generic methods can be either static or non-static.

Generic Method

```
class GenMethDemo {
// Determine if an object is in an array.
static <T, V extends T> boolean isIn(T x, V[] y) {
   for(int i=0; i < y.length; i++)
      if(x.equals(y[i])) return true;
      return false;
   }
public static void main(String args[]) {
   Integer nums[] = { 1, 2, 3, 4, 5 };
   if(isIn(2, nums))
      System.out.println("2 is in nums");

   String strs[] = {"one","two","three","four","five"};
   if(!isIn("seven", strs))
      System.out.println("seven is not in strs");
   }
}
-------------------------------------------------------
2 is in nums
seven is not in strs
```

Generic Constructor

```
class GenCons {
private double val;
<T extends Number> GenCons(T arg) {
   val = arg.doubleValue();
   }
void showval() {
   System.out.println("val: " + val);
   }
}
class GenConsDemo {
public static void main(String args[]) {
   GenCons test = new GenCons(100);
   GenCons test2 = new GenCons(123.5F);
   test.showval();
   test2.showval();
   }
}
-------------------------------------------------------
val: 100.0
val: 123.5
```

Inheritance in Generics

- Generic classes can be part of a class hierarchy
- In a generic hierarchy, any type arguments *needed* by a generic superclass *must be passed up* the hierarchy by all subclasses
- This is similar to the way that constructor arguments must be passed up a hierarchy.

Inheritance in Generics

```
class Gen<T> {
T ob;
Gen(T o) {
   ob = o;
   }
T getob() {
   return ob;
   }
}

class Gen2<T> extends Gen<T> {
Gen2(T o) {
   super(o);
   }
}
```

Inheritance in Generics

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

   Gen2<Integer> num = new Gen2<Integer>(100);
   System.out.print(num.getob());

   Gen2<String> str = new Gen2<String>(“Generics”);
   System.out.println(str.getob());
   }
}
-------------------------------------------------------
100
Generics
```

Inheritance in Generics

- Even if a subclass of a generic superclass would otherwise *not need to be generic*, it still must specify the type parameter(s) required by its generic superclass.
- A subclass is free to add its own type parameters, if needed.

Inheritance in Generics

```
class Gen<T> {
T ob; // declare an object of type T
Gen(T o) {
   ob = o;
   }
T getob() {
   return ob;
   }
}
class Gen2<T, V> extends Gen<T> {
V ob2;
Gen2(T o, V o2) {
   super(o);
   ob2 = o2;
   }
V getob2() {
   return ob2;
   }
}
```

Inheritance in Generics

```
class HierDemo {
public static void main(String args[]) {
   // Create a Gen2 object for String and Integer.
   Gen2<String, Integer> x =
   new Gen2<String, Integer>("Value is: ", 99);
   System.out.print(x.getob());
   System.out.println(x.getob2());
   }
}
```

```
---------------------------------------------------
Output:
Value is: 99
```

Generic Subclass

```
class NonGen {    // A non-generic class.
int num;
NonGen(int i) {
   num = i;
   }
int getnum() {
   return num;
   }
}

class Gen<T> extends NonGen {
T ob; // declare an object of type T
Gen(T o, int i) {
   super(i);
   ob = o;
   }
T getob() {
   return ob;
   }
}
```

Generic Subclass

```
class HierDemo2 {
public static void main(String args[]) {
   // Create a Gen object for String.
   Gen<String> w = new Gen<String>("Hello", 47);
   System.out.print(w.getob() + " ");
   System.out.println(w.getnum());
   }
}

-----------------------------------------------------
```

Output:

```
Hello 47
```

Casting

- Cast one instance of a generic class into another only if the two are compatible and their type arguments are the same.

```
(Gen<String>) w   //legal

(Gen<Long>) w  //illegal
```

Method overriding in Generics

```
class Gen<T> {
T ob;    // declare an object of type T
Gen(T o) {
   ob = o;
   }
T getob() {
   System.out.print("Gen's getob(): " );
   return ob;
   }
}
class Gen2<T> extends Gen<T> {
Gen2(T o) {
   super(o);
   }
// Override getob().
T getob() {
   System.out.print("Gen2's getob(): ");
   return ob;
   }
}
```

Method overriding in Generics

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

   Gen<Integer> iOb = new Gen<Integer>(88);
   Gen2<Integer> iOb2 = new Gen2<Integer>(99);
   Gen2<String> strOb2 = new Gen2<String> ("Generics
                                                  Test");

   System.out.println(iOb.getob());
   System.out.println(iOb2.getob());
   System.out.println(strOb2.getob());
   }
}
-----------------------------------------------------
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```