TOPIC 1

What Are Generics?

The term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, 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*.

Code that uses generics has the following benefits over non-generic code:

- **Type safety.**
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- **Elimination of casts.**
  The following code snippet without generics requires casting:
  - List list = new ArrayList();
  - list.add("hello");
  - String s = **(String)** list.get(0);

  When re-written to use generics, the code does not require casting:

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

- **Enabling programmers to implement generic algorithms.**
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generics add stability to your code by making more of your bugs detectable at compile time

A Simple Generics Example
The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```
// A simple generic class.Here, T is a type parameter that will be replaced by a real type when an object of type Gen is created.

class Gen<T> {
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o) {
ob = o;
}
// Return ob.
T getob() {
return ob;
}
// Show type of T.
void showType() {
System.out.println("Type of T is " +
ob.getClass().getName());
}
}
// Demonstrate the generic class.
class GenDemo {
public static void main(String args[]) {
// Create a Gen reference for Integers.
Gen<Integer> iOb;
// Create a Gen<Integer> object and assign its reference to iOb. Notice the use of autoboxing to encapsulate the value 88 within an Integer object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that
// no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String>("Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
}
}
```

First, notice how **Gen** is declared by the following line:

class Gen<T>

Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within < >. This syntax can be generalized. Whenever a type

parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type*

The **GenDemo** class demonstrates the generic **Gen** class. It first creates a version of **Gen** for integers, as shown here:

Gen<Integer> iOb;

Look closely at this declaration. First, notice that the type **Integer** is specified within the angle brackets after **Gen**. In this case, **Integer** is a *type argument* that is passed to **Gen**'s type parameter, **T**. This effectively creates a version of **Gen** in which all references to **T** are translated into references to **Integer**. Thus, for this declaration, **ob** is of type **Integer**, and the return type of **getob( )** is of type **Integer**.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name<type-param-list>* { // ...

Here is the syntax for declaring a reference to a generic class:

***class-name<type-arg-list> var-name* =new *class-name<type-arg-list>*(*cons-arg-list*);**

TOPIC II

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type. parameters: T and V.
class TwoGen<T, V> {
T ob1;
V ob2;
// Pass the constructor a reference to
// an object of type T and an object of type V.
TwoGen(T o1, V o2) {
ob1 = o1;
ob2 = o2;
}
// Show types of T and V.
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;
}
}
```

```
// Demonstrate TwoGen.
class SimpGen {
public static void main(String args[]) {
TwoGen<Integer, String> tgObj =new TwoGen<Integer, String>(88, "Generics");
// Show the types.
tgObj.showTypes();
// Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
String str = tgObj.getob2();
System.out.println("value: " + str);
}
}
```

Notice how **TwoGen** is declared:

class TwoGen<T, V> {

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created, as shown next:

TwoGen<Integer, String> tgObj =new TwoGen<Integer, String>(88, "Generics");

In this case, **Integer** is substituted for **T**, and **String** is substituted for **V**. Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

TwoGen<String, String> x = new TwoGen<String, String>("A", "B");

In this case, both **T** and **V** would be of type **String**.

TOPIC III
Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what *bounded type parameters* are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword , followed by its *upper bound* as shown here:

<*T* extends *superclass*>

This specifies that *T* can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```
class Stats<T > {
T[] nums; // array of Number or subclass

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;
}
}
// Demonstrate Stats.
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);
        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);
.}
}
```

In **Stats**, the **average( )** method attempts to obtain the **double** version of each number in the **nums** array by calling **doubleValue( )**. Because all numeric classes, such as **Integer** and **Double**, are subclasses of **Number**, and **Number** defines the **doubleValue( )** method, this method is available to all numeric wrapper classes. However the trouble is that the compiler has no way to know that you are intending to create **Stats** objects using only numeric types. Thus, when you try to compile **Stats**, an error is reported that indicates that the **doubleValue( )** method is unknown. To solve this problem, Java provides *bounded types*.

You can use an upper bound to fix the **Stats** class by specifying **Number** as an upper bound, as shown here:

```
class Stats<T extends Number> {
T[] nums; // array of Number or subclass

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;
}}
```

```
// Demonstrate Stats.
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);
        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);
.}
}
```

Because the type **T** is now bounded by **Number**, the Java compiler knows that all objects of type **T** can call **doubleValue( )** because it is a method declared by **Number**.

TOPIC IV
Using Wild card arguments

In generic code, the question mark (?), called the *wildcard*, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type. In all the situations mentioned above wild card must be used with generic class.

For example, you want to write the program to determines if two **Stats** objects contain arrays that yield the same average with the method name say **sameAvg()**, no matter what type of numeric data each object holds.

For example, if one object contains the **double** values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement **sameAvg( )** is to pass it a **Stats** argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same. For example, you want to be able to call **sameAvg( )**, as shown here:

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
System.out.println("Averages are the same.");
else
System.out.println("Averages differ.");
```

To create a generic sameAvg( ) method, you must use another feature of Java generics:

The *wildcard* argument. The wildcard argument is specified by the **?**, and it represents an unknown type. Using a wildcard, here is one way to write the **sameAvg( )** method:

```
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
if(average() == ob.average())
return true;
return false;
}
```

Here, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared.

The following program demonstrates this:

```
// Use a wildcard.
class Stats<T extends Number> {
T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o) {
        nums = o;
}
// Return type double in all cases.
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.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
    return true;
    return false;
}
}
// Demonstrate wildcard.
class WildcardDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);

        if(iob.sameAvg(dob))
            System.out.println("are the same.");
        else
            System.out.println("differ.");
}
}
```

It is important to understand that the wildcard does not affect what type of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration. The wildcard simply matches any *valid* **Stats** object.

Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type

1)Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its *upper bound*. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

The following is the illustration for Upper Bounded Wildcards

```
class UpperBoundWildDemo

{
public static double sumOfList(ArrayList<? extends Number> list) {
   double s = 0.0;
   for (Number n : list)
      s += n.doubleValue();
   return s;
}
public static void main(String[] args)
{
            ArrayList<Integer> al=new ArrayList<Integer>();

           al.add(10);
           al.add(20);
           al.add(30);
            System.out.println(UpperBoundWildDemo.sumOfList(al));
            ArrayList<Double> ald=new ArrayList<Double>();

           ald.add(10);
           ald.add(20);
           ald.add(30);
           System.out.println(UpperBoundWildDemo.sumOfList(ald));
}
}
```

2)Lower Bounded Wildcards

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 A>.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on ArrayList<Integer>, ArrayList<Number>, and ArrayList<Object> — anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify ArrayList<? super Integer>. The term ArrayList<Integer>is more restrictive than ArrayList<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

The following code adds the numbers 1 through 10 to the end of a list:

```
Class WildCardLowerBoundDemo
{
public static void addNumbers(ArrayList<? super Integer> list) {
   for (int i = 1; i <= 10; i++) {
      list.add(i);
   }
}
public static void main(String[] args)
{
            ArrayList<Integer> al=new ArrayList<Integer>();

            ArrayList<Number> aln=new ArrayList<Number>();
           ArrayList<Object> alo=new ArrayList<Object>();


           IldCardLowerBoundDemo. addNumbers(al);
           IldCardLowerBoundDemo. addNumbers(aln);
           IldCardLowerBoundDemo. addNumbers(alo);
}
}
```

Java Generics Wildcards usage is governed by the GET-PUT Principle . This states that: **Use an "extends" wildcard when you only get values out of a structure, Use a "super" wildcard when you only put values into a structure, and do not use wildcards when you do both.**

TOPIC V

Creating a Generic Method

It is possible to create a generic method that is enclosed within a non-generic class. Here is the syntax for a generic method:

<type-param-list> ret-type meth-name(param-list) { // ...

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Example:

The following program declares a non-generic class called **GenMethDemo** and a generic method within that class called **isIn( )**. The **isIn( )** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {
// Determine if an object is in an array.

 <T extends Comparable<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[]) {
        // Use isIn() on Integers.
        Integer nums[] = { 1, 2, 3, 4, 5 };
        if(isIn(2, nums))
                System.out.println("2 is in nums");
        if(!isIn(7, nums))
                System.out.println("7 is not in nums");
                System.out.println();

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

Let's examine **isIn( )** closely. First, notice how it is declared by this line:

<T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y)

The type parameters are declared *before* the return type of the method. Also note that **T** extends **Comparable<T>**. **Comparable** is an interface declared in **java.lang.** A class that implements **Comparable** defines objects that can be ordered. Thus, requiring an upper bound of **Comparable** ensures that **isIn**( ) can be used only with objects that are capable of being compared. **Comparable** is generic, and its type parameter specifies the type of objects that it compares. Next, notice that the type **V** is upper-bounded by **T**. Thus, **V** must either be the same as type **T**, or a subclass of **T**. This relationship enforces that **isIn( )** can be called only with arguments that are compatible with each other

TOPIC VI

Generic Constructors

It is possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```
// Use a 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();
}
}
```

Because **GenCons( )** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons( )** can be called with any numeric type, including **Integer**, **Float**, or **Double**. Therefore, even though **GenCons** is not a generic class, its constructor is generic.

The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented. In the **MinMax** example, only types that implement the **Comparable** interface can be passed to **T**.

TOPIC VII

Generic Interfaces

Generic interfaces are specified just like generic classes. Here is the generalized syntax for a generic interface:

interface *interface-name<type-param-list>* { // ...

Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:

class *class-name<type-param-list>* implements *interface-name<type-arg-list>* {

Example:

It creates an interface called **MinMax** that declares the methods **min( )** and **max( )**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.
// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
        T min();
        T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
        T[] vals;
        MyClass(T[] o) { vals = o; }
        // Return the minimum value in vals.
        public T min() {
                T v = vals[0];
                for(int i=1; i < vals.length; i++)
                if(vals[i].compareTo(v) < 0) v = vals[i];
                return v;
}
// Return the maximum value in vals.
public T max() {
        T v = vals[0];
        for(int i=1; i < vals.length; i++)
                if(vals[i].compareTo(v) > 0) v = vals[i];
                        return v;
}
}
class GenIFDemo {
        public static void main(String args[]) {
                Integer inums[] = {3, 6, 2, 8, 6 };
                Character chs[] = {'b', 'r', 'p', 'w' };
                MyClass<Integer> iob = new MyClass<Integer>(inums);
                MyClass<Character> cob = new MyClass<Character>(chs);
                System.out.println("Max value in inums: " + iob.max());
                System.out.println("Min value in inums: " + iob.min());
                System.out.println("Max value in chs: " + cob.max());
                System.out.println("Min value in chs: " + cob.min());
  }
}
```

TOPIC VIII

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class.Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that 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.

1.Using a Generic Superclass

Here is a simple example of a hierarchy that uses a generic superclass:

```
// A simple generic class hierarchy.
class Gen<T> {
        T ob;
        Gen(T o) {
                ob = o;
        }
        // Return ob.
        T getob() {
                return ob;
        }
}
// A subclass of Gen.
class Gen2<T> extends Gen<T> {
        Gen2(T o) {
                super(o);
        }
}
```

In this hierarchy, **Gen2** extends the generic class **Gen**. Notice how **Gen2** is declared by the following line:

class Gen2<T> extends Gen<T> {

The type parameter **T** is specified by **Gen2** and is also passed to **Gen** in the **extends** clause. This means that whatever type is passed to **Gen2** will also be passed to **Gen**. For example, the following declaration,

Gen2<Integer> num = new Gen2<Integer>(100);

passes **Integer** as the type parameter to **Gen**. Thus, the **ob** inside the **Gen** portion of **Gen2** will be of type **Integer**.

Notice also that **Gen2** does not use the type parameter **T** except to pass it to the **Gen** superclass. Thus, 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. For example, here is a variation on the preceding hierarchy in which **Gen2** adds a type parameter of its own:

```
// A subclass can add its own type parameters.
class Gen<T> {
        T ob; // declare an object of type T
        // Pass the constructor a reference to an object of type T.
        Gen(T o) {
        ob = o;
}
        // Return ob.
        T getob() {
                return ob;
        }
}
// A subclass of Gen that defines a second type parameter, called V.
class Gen2<T, V> extends Gen<T> {
        V ob2;
        Gen2(T o, V o2) {
                super(o);
                ob2 = o2;
        }
        V getob2() {
                return ob2;
        }
}
// Create an object of type Gen2.
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());
        }
}
```

Notice the declaration of this version of **Gen2**, which is shown here:

class Gen2<T, V> extends Gen<T> {

Here, **T** is the type passed to **Gen**, and **V** is the type that is specific to **Gen2**. **V** is used to declare an object called **ob2**, and as a return type for the method **getob2( )**. In **main( )**, a **Gen2** object is created in which type parameter **T** is **String**, and type parameter **V** is **Integer**.

2.A Generic Subclass

A non-generic class can be the superclass of a generic subclass.

For example, consider this program:

```
class NonGen {
                int num;
                NonGen(int i) {
                num = i;
}
                int getnum() {
                                return num;
                }
}
// A generic subclass.
class Gen<T> extends NonGen {
                T ob; // declare an object of type T
                // Pass the constructor a reference to an object of type T.
                Gen(T o, int i) {
                        super(i);
                        ob = o;
                }
                // Return ob.
                T getob() {
                return ob;
        }
}
// Create a Gen object.
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());
        }
}
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

class Gen<T> extends NonGen {

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way.

TOPIC IX

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

To better understand how erasure works, consider the following class:

```
// Here, T is bounded by Object i.e. java.lang.Object
class Gen<T> {
// Here, T will be replaced by default i.e. Object
   T obj;

   Gen(T o)
   {
      obj = o;
   }
   T getob()
   {
      return obj;
   }
}
```

After compilation, the code is replaced by default Object like the below:

```
class Gen
{
// Here, T will be replaced by default i.e. Object
   Object obj;
   Gen(Object o)
   {
      obj=o;
   }
   Object getob()
   {
      return obj;
   }
}
```

After the compilation, in Gen class T will be replaced by Object and in Geeks class T will be replaced by String. We can check the content by running the compiled code by **javap className** command.

TOPIC X
Ambiguity Errors

The inclusion of generics gives rise to a new type of error that you must guard against: ambiguity.

Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on overloaded methods.

class MyGenClass<T, V> {
        T ob1;
        V ob2;
        // ...
        // These two overloaded methods are ambiguous and will not compile.
        void set(T o) {
                ob1 = o;
        }
        void set(V o) {
        ob2 = o;
        }
}
```

Notice that MyGenClass declares two generic types: T and V. Inside MyGenClass, an attempt is made to overload set( ) based on parameters of type T and V. This looks reasonable because T and V appear to be different types. However, there are two ambiguity problems here.

First, as MyGenClass is written, there is no requirement that T and V actually be different types. For example, it is perfectly correct (in principle) to construct a MyGenClass object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both T and V will be replaced by String. This makes both versions of set( ) identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of set( ) reduces both versions to the following:

```
void set(Object o) { // ...
```

Thus, the overloading of set( ) as attempted in MyGenClass is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that V will always be some type of Number, you might try to fix MyGenClass by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!
```

This change causes MyGenClass to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

In this case, since both T and V are Number, which version of set( ) is to be called? **The call to set( ) is now ambiguous.**

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload set( ). Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

TOPIC XI
Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

1.Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
        T ob;
        Gen() {
                ob = new T(); // Illegal!!!
        }
}
```

Here, it is illegal to attempt to create an instance of T.

2.Restrictions on Static Members

No static member can use a type parameter declared by the enclosing class. For example, all of the static members of this class are illegal:

```
class Wrong<T> {
        // Wrong, no static variables of type T.
        static T ob;

        // Wrong, no static method can use T.
        static T getob() {
                return ob;
        }
        // Wrong, no static method can access object of type T.
        static void showob() {
                System.out.println(ob);
       }
}
```

Although you can't declare static members that use a type parameter declared by the enclosing class, you can declare static generic methods, which define their own type parameters.

3.Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose base type is a type parameter. Second, you cannot create an array of type specific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
        T ob;
        T vals[]; // OK
        Gen(T o, T[] nums) {
                ob = o;
        // This statement is illegal.
        // vals = new T[10]; // can't create an array of T
        // But, this statement is OK.
        vals = nums; // OK to assign reference to existent array
        }
}
class GenArrays {
        public static void main(String args[]) {
                Integer n[] = { 1, 2, 3, 4, 5 };
                Gen<Integer> iOb = new Gen<Integer>(50, n);
                // Can't create an array of type-specific generic references.
                // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
                // This is OK.
                Gen<?> gens[] = new Gen<?>[10]; // OK
        }
}
```

As the program shows, it's valid to declare a reference to an array of type T, as this line does:
T vals[]; // OK
But, you cannot instantiate an array of T, as this commented-out line attempts:
// vals = new T[10]; // can't create an array of T

The reason you can't create an array of T is that T does not exist at run time, so there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to Gen( ) when an object is created and assign that reference to vals, as the program does in this line:
vals = nums; // OK to assign reference to existent array

This works because the array passed to Gen has a known type, which will be the same type as T at the time of object creation.

Inside main( ), notice that you can't declare an array of references to a specific generic type. That is, this line

// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

won’t compile. Arrays of specific generic types simply aren’t allowed, because they can lead to a loss of type safety.

You can create an array of references to a generic type if you use a wildcard, however, as shown here:

Gen<?> gens[] = new Gen<?>[10]; // OK

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

4. Generic Exception Restriction

Ageneric class cannot extend Throwable. This means that you cannot create generic exception classes.