Generics

What Are Generics?

- At its core, 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.

A Simple Generics Example

- Let’s begin with a simple example of a generic class. The following program defines two classes. The first is the generic class Gen, and the second is GenDemo, which uses Gen.

```java
// 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); } }
```

Java

- 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.

```java
iOb = new Gen<Double>(88.0); // Error!
```

Java

- Because iOb is of type Gen<Integer>, it can't be used to refer to an object of Gen<Double>. This type checking is one of the main benefits of generics because it ensures type safety.

Generics Work Only with Reference Types

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as int or char. For example, with Gen, it is possible to pass any class type to T, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

```java
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

Java

Generic Types Differ Based on Their Type Arguments

- A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

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

Java

- Even though both iOb and strOb are of type Gen<T>, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

A generic class with two type parameters

```
listing 3 // 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. 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); } }
```

Java

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 > { // ...
```

Java

- Here is the full syntax for declaring a reference to a generic class and instance creation:

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

Java

Bounded Types

- In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. 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

```
listing 4 // Stats attempts (unsuccessfully) to // create a generic
class that can compute // the average of an array of numbers of //
any given type. // // The class contains an error! class Stats<T> {
T[] nums; // nums is an array of type T // Pass the constructor a
reference to // an array of type T. 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(); //
Error!!! return sum / nums.length; } }
```

Java

- To handle such situations, Java provides bounded types. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:

```
<T extends superclass>
```

Java

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

```java
// In this version of Stats, the type argument for // T must be either Number, or a class derived // from Number. 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; } } // 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); // This won't compile because String is not a // subclass of Number. // String strs[] = { "1", "2", "3", "4", "5" }; // Stats<String> strob = new Stats<String>(strs); // double x = strob.average(); // System.out.println("strob average is " + v); } }
```

Java

- In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. When specifying a bound that has a class and an interface, or multiple interfaces, use the & operator to connect them. For example,

```java
class Gen<T extends MyClass & MyInterface> { // ...
```

Java

- Here, T is bounded by a class called MyClass and an interface called MyInterface. Thus, any type argument passed to T must be a subclass of MyClass and implement MyInterface.