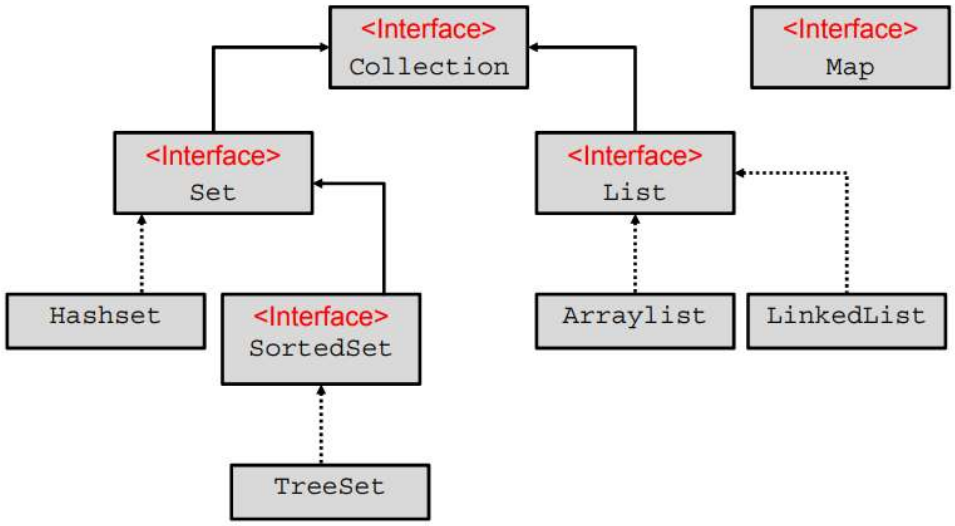
Collections and Generics

Collection Fundamentals

The Java platform includes a collections framework. A **collection** is an object that represents a group of objects. A collection framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

- Objects in a collection are called **elements**.
- The Collections API relies heavily on generics for its implementation.
- The Collections classes are all stored in the java.util package.



The diagram in the slide shows the Collection framework. The framework is made up of a set of interfaces for working with a group (collection) of objects.

Collection Interfaces and Implementation

Interface	Implementation		
List	ArrayList	LinkedList	
Set	TreeSet	HashSet	LinkedHashSet
Map	HashMap	HashTable	TreeMap
Deque	ArrayDeque		

List Interface

- List defines generic list behaviour.
    - Is an ordered collection of elements.
- List behaviours include
    - Adding elements at a specific index.
    - Getting an element based on an index.
    - Removing an element based on an index.
    - Overwriting an element based on an index.
    - Getting the size of the list.
- List allows duplicate elements.

ArrayList

- Is an implementation of the List interface.
    - The list automatically grows if elements exceed initial size.
- Has numeric index.
    - Elements are accessed by index.
    - Elements can be inserted based on index.
    - Elements can be overwritten.
- Allows duplicate items.

[Example in codes]

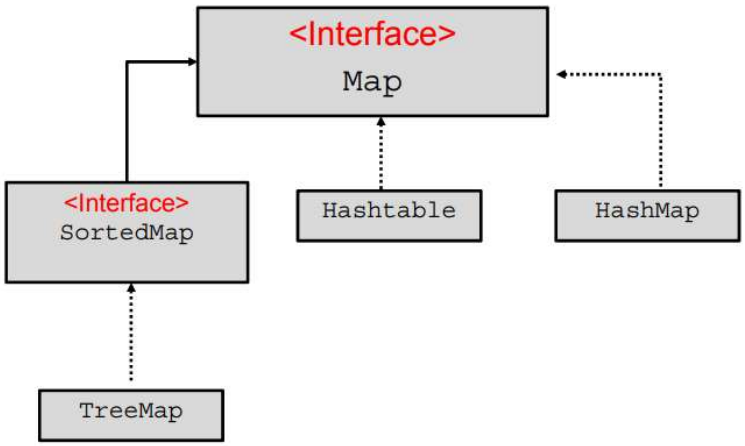
Set Interface

- A Set in an interface that contains only unique elements.
- A Set has no index.
- Duplicate elements are not allowed.
- You can iterate through elements to access them.
- TreeSet provides sorted implementation.

[Example in codes]

Map Interface

- A collection that stores multiple key-value pairs.
    - Key: unique identifier for each element in a collection.
    - Value: a value stored in the element associated with the key.
- Called "associative arrays" in other language.



The Map interface does not extend the Collection interface because it represents mappings and not a collection of objects. Some of the key implementation classes includes:

- TreeMap: a map where keys are automatically sorted.
- Hashtable: a classic associative array implementation with keys and values. Hashtable is synchronized.
- HashMap: an implementation just like Hashtable except that it accepts null keys and values. Also, it is not synchronized.

[Example in codes]

Ordering Collections

- The Comparable and Comparator interfaces are used to sort collections.
    - Both are implemented by using generics.
- Using Comparable interface.
    - Overrides the compareTo method.
    - Provides only one sort option.
- The Comparator interface.
    - Is implemented by using the compare method.
    - Enables you to create multiple Comparator classes.
    - Enables you to create and use numerous sorting options.

[Example in codes]

Generic Fundamentals

Since the original 1.0 release in 1995, many new features have been added to Java. One that has had a profound impact is generics. Introduced by JDK 5, generics changed Java in two important ways. First, it added a new syntactical element to the language. Second, it caused changes to many of the classes and methods in the core API. Today, generics are an integral part of Java programming, and a solid understanding of this important feature is required.

- The term generics means parameterized types.
- 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.
- Generics added the type safety.
- With generics, all casts are automatic and implicit.

A Simple Generic Example

```
//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.
public class Test {
	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);
	}
}

/*
 * The output produced by the program is shown here:
	Type of T is java.lang.Integer
	value: 88
	Type of T is java.lang.String
	value: Generics Test
 */
```

Bounded Types

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. Thus, you want to specify the type of the numbers generically, using a type parameter.

```
//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.
public class Test {
	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);
	}
}

/*
 * The output is shown here:
	Average is 3.0
	Average is 3.3
 */
```

Using Wildcard Arguments

```
//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.
public class Test {
	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);
		// 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.");
	}
```

```
}

/*
 * The output is shown here:
        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.
 */
```

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

One last point: 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.

Consider the following hierarchy of classes that encapsulate coordinates:

```
//Two-dimensional coordinates.
class TwoD {
        int x, y;

        TwoD(int a, int b) {
                x = a;
                y = b;
        }
}

//Three-dimensional coordinates.
class ThreeD extends TwoD {
        int z;

        ThreeD(int a, int b, int c) {
                super(a, b);
                z = c;
        }
}

//Four-dimensional coordinates.
class FourD extends ThreeD {
        int t;

        FourD(int a, int b, int c, int d) {
```

```
                super(a, b, c);
                t = d;
        }
}
```

At the top of the hierarchy is TwoD, which encapsulates a two-dimensional, XY coordinate. TwoD is inherited by ThreeD, which adds a third dimension, creating an XYZ coordinate. ThreeD is inherited by FourD, which adds a fourth dimension (time), yieldinga four-dimensional coordinate.

Shown next is a generic class called Coords, which stores an array of coordinates:

```
//This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
        T[] coords;

        Coords(T[] o) {
                coords = o;
        }
}
```

Notice that Coords specifies a type parameter bounded by TwoD. This means that any array stored in a Coords object will contain objects of type TwoD or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the coords array of a Coords object. Because all types of Coords objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```
static void showXY(Coords<?> c) {
        System.out.println("X Y Coordinates:");
        for (int i = 0; i < c.coords.length; i++)
                System.out.println(c.coords[i].x + " " + c.coords[i].y);
        System.out.println();
}
```

Because Coords is a bounded generic type that specifies TwoD as an upper bound, all objects that can be used to create a Coords object will be arrays of type TwoD, or of classes derived from TwoD. Thus, showXY( ) can display the contents of any Coords object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a ThreeD or FourD object? The trouble is that not all Coords objects will have three coordinates, because a Coords<TwoD> object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for Coords<ThreeD> and

Coords<FourD> objects, while preventing that method from being used with Coords<TwoD> objects? The answer is the bounded wildcard argument.

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.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a Coords object, if that object actually has those three coordinates. For example, the following showXYZ( ) method shows the X, Y, and Z coordinates of the elements stored in a Coords object, if those elements are actually of type ThreeD (or are derived from ThreeD):

```
static void showXYZ(Coords<? extends ThreeD> c) {
        System.out.println("X Y Z Coordinates:");
        for (int i = 0; i < c.coords.length; i++)
                System.out.println(c.coords[i].x + " " + c.coords[i].y + " " + c.coords[i].z);
        System.out.println();
}
```

Generic Methods

The following program declares a non-generic class called GenMethDemo and a static 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.
public class Test {

	// Determine if an object is in an array.
	static <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");

		// Oops! Won't compile! Types must be compatible.
		// if(isIn("two", nums))
		// System.out.println("two is in strs");
	}
}

/*
 * The output from the program is shown here:
	2 is in nums
	7 is not in nums
 */
```

Generic Constructors

It is possible for constructors to be generic, even if their class is not.

```
//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);
	}
}

public class Test {

	public static void main(String args[]) {
		GenCons test = new GenCons(100);
		GenCons test2 = new GenCons(123.5F);
		test.showval();
		test2.showval();
	}
}

/*
 * The output is shown here:
	val: 100.0
	val: 123.5
 */
```

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an 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;
	}
}

public class Test {

	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());
	}
```

}

Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code. To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a raw type for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost. Here is an example that shows a raw type in action:

```
//Demonstrate a raw type.
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;
        }
}

//Demonstrate raw type.
public class Test {

        public static void main(String args[]) {
                // Create a Gen object for Integers.
                Gen<Integer> iOb = new Gen<Integer>(88);
                // Create a Gen object for Strings.
                Gen<String> strOb = new Gen<String>("Generics Test");
                // Create a raw-type Gen object and give it
                // a Double value.
                Gen raw = new Gen(new Double(98.6));
                // Cast here is necessary because type is unknown.
                double d = (Double) raw.getob();
                System.out.println("value: " + d);

                // The use of a raw type can lead to run-time
                // exceptions. Here are some examples.

                // The following cast causes a run-time error!
                // int i = (Integer) raw.getob(); // run-time error

                // This assignment overrides type safety.
                strOb = raw; // OK, but potentially wrong
                // String str = strOb.getob(); // run-time error

                // This assignment also overrides type safety.
                raw = iOb; // OK, but potentially wrong
```

```
                // d = (Double) raw.getob(); // run-time error
        }
}
```

Type Inference with the Diamond Operator

Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
        T ob1;
        V ob2;

        MyClass(T o1, V o2) {
                ob1 = o1;
                ob2 = o2;
        }
// ...
}
```

Prior to JDK 7, to create an instance of MyClass, you would have needed to use a statement similar to the following:

```
MyClass<Integer, String> mcOb = new MyClass<Integer, String>(98, "A String");
```

Today the preceding declaration can be rewritten as shown here:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

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.

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. The reason should be easy to understand: the compiler does not know what type of object to create. T is simply a placeholder.

Restrictions on Static Members

No static member can use a type parameter declared by the enclosing class. For example, both 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;
        }
}
```

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.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element 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
        }
}

public class Test {
        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
        }
}
```

Generic Exception Restriction

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

Using Java's Type Wrappers to Convert Numeric to Strings

Sometimes, you need to convert a number to a string because you need to operate on the value in its string form. All classes inherit a method called toString from the Object class. The type-wrapper classes override this method to provide a reasonable string representation of the value held by the number object.

The following program, ToStringDemo (in a .java source file), uses the toString method to convert a number to a string. Next, the program uses some string methods to compute the number of digits before and after the decimal point:

```
double d = 858.48;
String s = Double.toString(d);

int dot = s.indexOf('.');
System.out.println(s.substring(0, dot).length() + " digits before decimal point.");
System.out.println(s.substring(dot + 1).length() + " digits after decimal point.");
```

The toString method called by this program is the class method. Each of the number classes has an instance method called toString, which you call on an instance of that type.

You don't have to explicitly call the toString method to display numbers with the System.out.println method or when concatenating numeric values to a string. The Java platform handles the conversion by calling toString implicitly.

List of wrapper classes.

Byte	Short	Integer	Long
Float	Double	Character	Boolean