

```
Collection<?> collUnknown;
```

Note that we can not add anything to this collection. We can only take out elements of type Object from it. So what is the use of this variable if we can not add anything to the collection it represents? The use of this new construct will be clear when you want to create a generic method that takes any collection.

```
public static void printElements( Collection<?> anycoll )
{
    Iterator<?> iter = coll.iterator();
    while ( iter.hasNext() )
    {
        System.out.print( iter.next() );
    }
}
```

Wildcard for a specific type of classes

"<? extends ClassName>" specifies a restriction on the types of classes that may be used.

For example, to create a collection that may only contain "Serializable" objects, specify:

```
Collection<? extends Serializable> serColl = new
ArrayList<String>();
```

The above code is valid because, the String class is serializable. Use of a class that is not serializable would cause a compilation error.

The following collection can only contain objects that extend the class Animal.

```
class Dog extends Animal
{
    ...
}
...
// --- Create "Animal Collection" variable ---
Collection<? extends Animal> animalColl = new
ArrayList<Dog>();
```

"<? super ClassName>" specifies a restriction on the types of classes that may be used.

For example, to declare a Comparator that can compare Dogs, you use

```
Comparator<? super Dog> myComparator;
```

Now suppose you define a comparator that can compare Animals:

```
class AnimalComparator extends Comparator<Animal>
{
    int compare(Animal a, Animal b) { //...
    }
}
```

Since Dogs are Animals, you can use this comparator to compare Dogs also. Comparators for any superclass of Dog can also compare Dog; but comparators for any strict subclass cannot.

```
Comparator<Animal> myAnimalComparator = new AnimalComparator();

static int compareTwoDogs(Comparator<? super Dog> comp, Dog dog1,
    Dog dog2) {
    return comp.compare(dog1, dog2);
}
```

The above code is valid because, the class is a supertype of the class. Use of a class that is not a supertype would cause a compilation error.

Defining Classes

Navigate **Classes and Objects** topic:

- Defining classes
- Creating objects
- Interfaces
- Using static members
- Destroying objects
- Overloading methods and constructors
- Class Loading

Fundamentals

Every class in Java can be composed of the following elements

- **fields** - Fields are variables that hold data specific to each object. For example, an employee might have an ID number. (They are also called member variables.) There is one field for each object of a class.
- **member methods** - Member methods perform operations on an object. For example, an employee might have a method to issue his paycheck or to access his name.
- **static fields** - Static fields are common to any object of the same class. For example, a static field within the Employee class could keep track of the last ID number issued. Only one static field exists for one class.
- **static methods** - Static methods are methods that do not affect a specific object.
- **other classes** - Sometimes it is useful to contain a class within another one if it is useless outside of the class or should not be accessed outside the class.
- **Constructors** - A special method that generates a new object.
- **Parameterized types** - Since 1.5, 'parameterized types' can be assigned to a class during definition. The 'parameterized types' will be substituted with the types specified at the class's instantiation. It is done by the compiler. It is similar to the C language macro '#define' statement, where a preprocessor evaluates the macros.

```
public class Employee // This defines the Employee
class.
{ // The public modifier indicates
that // it can be accessed by any other
class
    private static int nextID; // Define a static field. Only one
copy of this will exist, // no matter how many Employees
```

are created.

```

    private int myID;           // Define fields that will be
stored
    private String myName;      // for each Employee. The private
modifier indicates that
                                // only code inside the Employee
class can access it.

    public Employee(String name) // This is a constructor. You can
pass a name to the constructor
    {
        created Employee object.
        myName = name;
        myID = nextID;         // Automatically assign an ID to
the object
        nextID++;              // Increment the ID counter
    }

    public String getName()     // This is a member method that
returns the
    {
        // Employee object's name.
        return myName;         // Note how it can access the
private field myName.
    }

    public int getID()          // This is another member method.
    {
        return myID;
    }

    public static int getNextID() // This is a static method that
returns the next ID
    {
        // that will be assigned if
another Employee is created.
        return nextID;
    }
}

```

The following Java code

```

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

        System.out.println(Employee.getNextID());

        Employee a = new Employee("John Doe");
        Employee b = new Employee("Jane Smith");
    }
}

```