Generic Programming
Unit 5

Compiled by
M S Anand

Department of Computer Science

Generic Programming

Text Book:

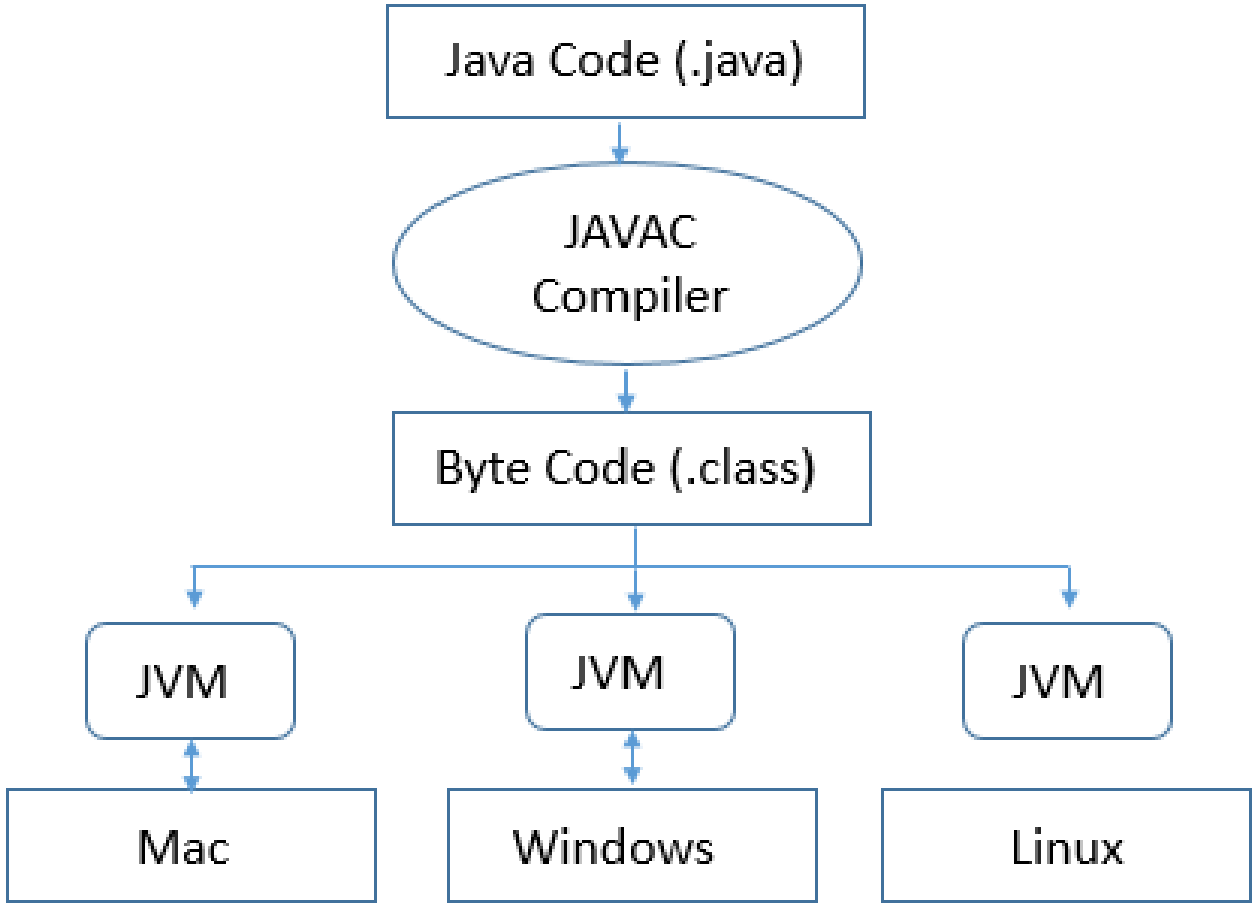
1: “STL Tutorial and Reference”, Musser, Derge and Saini, 2nd Edition, Addison-Wesley, 2001.

Reference Book(s):

1: “C++ Primer”, Lippman, Addison-Wesley, 2013.
2: “A tour of C++”, Bjarne Stroustrup, Addison-Wesley, 2013.
3: “Templates: The Complete Guide”, David Vandevoorde, Nicolai M Josuttis, Addison-Wesley, 2002.
4: “Java Tutorials”, Online Reference Link - https://docs.oracle.com/javase/tutor.
5: MSDN for C# generics.

Generic Programming

Java in a nutshell



Generic Programming

Java in a nutshell

The first Java program

```
public class MyFirstApp
{
    public static void main (String args[])
    {
        System.out.println (“Welcome to the world of Java”);
    }
}
```

Store it in a file called MyFirstApp.java

Compile the program from the command line”
javac MyFirstApp.java (This results in MyFirstApp.class file being created. This is the bytecode)

Execute the program:
java MyFirstApp

Passing Arguments to the main() Method

You can pass arguments from the command line to the main() method.
The main() method can access the arguments from the command line like this:

```
class Sample
{
    public static void main (String [] args)
   {
        args[0] – Contains the first argument
        args[1] – Contains the second argument
        .
        .
}
```

How do you know how many arguments have been passed?
args.length gives the number of command line arguments passed.

Keywords – Primitive data types

Keyword	Meaning
boolean	A data type that can hold either true or false
byte	A data type that can hold a 8-bit data value
char	A data type that can hold unsigned 16-bit Unicode characters
short	A data type that can hold a 16-bit integer
int	A data type that can hold a 32-bit signed integer
long	A data type that can hold a 64-bit integer
float	A data type that can hold a 32-bit floating point number
double	A data type that can hold a 64-bit floating point number
void	Specifies that a method does not have a return value

Generic Programming

Java in a nutshell

Modifiers

Keyword	Meaning
public	An access specifier used for classes, interfaces, methods and variables indicating that an item is accessible throughout the application
protected	An access specifier indicating that a method or a variable may only be accessed in the class it has declared (or a subclass of the class it has declared in or other classes in the same package)
private	An access specifier indicating that a method or variable may be accessed only in the class it's declared in.
abstract	Specifies that a class or method will be implemented later in a subclass.
static	Indicates that a variable or method is a class method (not limited to one object)
final	Indicates that a variable holds a constant value or that a method will not be overridden
transient	Indicates that a variable is not a part of an object's persistent state
volatile	Indicates that a variable may change asynchronously.
synchronized	Specifies critical sections or methods in a multithreaded code
native	Specifies that a method is implemented with native (platform-specific) code.

Declarations

Keyword	Meaning
class	It declares a new class
interface	It declares a new interface
enum	It declares enumerated type variables
extends	Indicates that a class is derived from another class or an interface is derived from another interface
implements	Specifies that a class implements an interface
package	Declares a Java package
throws	Indicates what exceptions may be thrown by a method

Generic Programming

Java in a nutshell

Primitive data types

Data type	Description
boolean	A binary value of either true or false
byte	8 bit signed value, values from -128 to 127
short	16 bit signed value, values from -32.768 to 32.767
char	16 bit Unicode character
int	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
float	32 bit floating point value
double	64 bit floating point value

Generic Programming

Java in a nutshell

Object Types

The primitive types also come in versions that are full-blown objects. That means that you reference them via an object reference.

Data type	Description
Boolean	A binary value of either true or false
Byte	8 bit signed value, values from -128 to 127
Short	16 bit signed value, values from -32.768 to 32.767
Character	16 bit Unicode character
Integer	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
Long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
Float	32 bit floating point value
Double	64 bit floating point value
String	N byte Unicode string of textual data. Immutable

Auto Boxing

Before Java 5, you had to call methods on the object versions of the primitive types, to get their value out as a primitive type. For instance:

```
Integer myInteger = new Integer(45);

int myInt = myInteger.intValue();
```

From Java 5, you have a concept called "auto boxing". That means that Java can automatically "box" a primitive variable in an object version, if that is required, or "unbox" an object version of the primitive data type if required. For instance, the example before could be written like this:

```
Integer myInteger = new Integer(45);
int myInt = myInteger;
```

In this case Java would automatically extract the int value from the myInteger object and assign that value to myInt.

Similarly, creating an object version of a primitive data type variable was a manual action before Java:

```
int myInt = 45;
Integer myInteger = new Integer(myInt);
```

With auto boxing Java can do this for you. Now you can write:

```
int myInt = 45;
Integer myInteger = myInt;
```

Java will then automatically "box" the primitive data type inside an object version of the corresponding type.

Java's auto boxing features enables you to use primitive data types where the object version of that data type was normally required, and vice versa.

Summary of Operators

Simple Assignment Operator

= Simple assignment operator

Arithmetic Operators

+ Additive operator (also used for String concatenation)
- Subtraction operator
* Multiplication operator
/ Division operator
% Remainder operator

Unary Operators

+ Unary plus operator; indicates positive value (numbers are positive without this, however)
- Unary minus operator; negates an expression
++ Increment operator; increments a value by 1
-- Decrement operator; decrements a value by 1
! Logical complement operator; inverts the value of a boolean

Generic Programming

Java in a nutshell

Equality and Relational Operators

==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Conditional Operators

&&	Conditional-AND
||	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

Type Comparison Operator

instanceof	Compares an object to a specified type

Bitwise and Bit Shift Operators

~	Unary bitwise complement		
<<	Signed left shift	>>	Signed right shift
>>>	Unsigned right shift		
&	Bitwise AND		
^	Bitwise exclusive OR	|	Bitwise inclusive OR

Java Arrays

Declaring an Array Variable in Java

int[] intArray; or int intArray[];

String[] stringArray; or String stringArray [];

MyClass[] myClassArray; or MyClass myClassArray[];

Instantiating an Array in Java

When you declare a Java array variable you only declare the variable (reference) to the array itself. The declaration does not actually create an array. You create an array like this:

```
int[] intArray;

intArray = new int[10];

String[] stringArray = new String[10];
```

Java Array Literals

The Java programming language contains a shortcut for instantiating arrays of primitive types and strings. If you already know what values to insert into the array, you can use an array literal.

```
int[]   ints2 = new int[]{ 1,2,3,4,5,6,7,8,9,10 };
```

Actually, you don't have to write the new int[] part in the latest versions of Java. You can just write:

```
int[]   ints2 = { 1,2,3,4,5,6,7,8,9,10 };
```

It is the part inside the curly brackets that is called an array literal.

Generic Programming
Java in a nutshell

String[] strings = {"one", "two", "three"};

Java Array Length Cannot Be Changed
Once an array has been created, its size cannot be resized.

Accessing Java Array Elements
Each variable in a Java array is also called an "element".

You can access each element in the array via its index (starts from 0).

Array Length
You can access the length of an array via its length field.

int[] intArray = new int[10];

int arrayLength = intArray.length;

Generic Programming
Java in a nutshell

Iterating Arrays

```
String[] stringArray = new String[10];

for(int i=0; i < stringArray.length; i++) {
    stringArray[i] = "String no " + i;
}
```

You can also iterate an array using the "**for-each**" loop in Java.

```
int[] intArray = new int[10];

for(int theInt : intArray) {
    System.out.println(theInt);
}
```

The for-each loop gives you access to each element in the array, one at a time, but gives you no information about the index of each element. Additionally, you only have access to the value. You cannot change the value of the element at that position. If you need that, use a normal for-loop as shown earlier.

Multidimensional Java Arrays

```
int[][] intArray = new int[10][20];
```

Inserting elements into an array

Sometimes you need to insert elements into a Java array somewhere. Here is how you insert a new value into an array in Java:

```
int[] ints   = new int[20];
int insertIndex = 10;
int newValue    = 123;
//move elements below insertion point.
for(int i=ints.length-1; i > insertIndex; i--){
    ints[i] = ints[i-1];
}
//insert new value
ints[insertIndex] = newValue;

System.out.println(Arrays.toString(ints));
```

Generic Programming

Java in a nutshell

The Arrays Class

Java contains a special utility class that makes it easier for you to perform many often used array operations like copying and sorting arrays, filling in data, searching in arrays etc. The utility class is called Arrays and is located in the standard Java package java.util. Thus, the fully qualified name of the class is:

java.util.Arrays

In order to use java.util.Arrays in your Java classes you must import it. Here is how importing java.util.Arrays could look in a Java class of your own:

```
package myjavaapp;

import java.util.Arrays;
```

Copying Arrays

Copying an Array by Iterating the Array

Copying an Array Using Arrays.copyOf()

The second method to copy a Java array is to use the Arrays.copyOf() method.

```
int[] source = new int[10];

for(int i=0; i < source.length; i++) {
    source[i] = i;
}

int[] dest = Arrays.copyOf(source, source.length);
```

The Arrays.copyOf() method takes 2 parameters.
The first parameter is the array to copy.
The second parameter is the length of the new array. This parameter can be used to specify how many elements from the source array to copy.

Generic Programming

Java in a nutshell

Sorting Arrays

You can sort the elements of an array using the **Arrays.sort()** method. Sorting the elements of an array rearranges the order of the elements according to their sort order.

Filling Arrays With **Arrays.fill()**

The Arrays class has set of methods called fill(). These Arrays.fill() methods can fill an array with a given value.

Searching Arrays with **Arrays.binarySearch()**

The Arrays class contains a set of methods called binarySearch(). This method helps you perform a binary search in an array. The array must first be sorted.

Check if Arrays are Equal with **Arrays.equals()**

Java Strings

The Java String data type can contain a sequence (string) of characters. Strings are how you work with text in Java. Once a Java String is created you can search inside it, create substrings from it, create new strings based on the first but with some parts replaced, plus many other things.

Creating a String

Strings in Java are objects. Therefore, you need to use the new operator to create a new Java String object.

```
String myString = new String("Hello World");
```

or

```
String myString = "Hello World“;
```

Generic Programming

Java in a nutshell

Java String library

public class String

	String(String s)	*create a string with the same value as* `s`
	String(char[] a)	*create a string that represents the same sequence of characters as in* `a[]`
int	length()	*number of characters*
char	charAt(int i)	*the character at index* `i`
String	substring(int i, int j)	*characters at indices* `i` *through* `(j-1)`
boolean	contains(String substring)	*does this string contain* `substring`*?*
boolean	startsWith(String prefix)	*does this string start with* `prefix`*?*
boolean	endsWith(String postfix)	*does this string end with* `postfix`*?*
int	indexOf(String pattern)	*index of first occurrence of* `pattern`
int	indexOf(String pattern, int i)	*index of first occurrence of* `pattern` *after* `i`
String	concat(String t)	*this string, with* `t` *appended*
int	compareTo(String t)	*string comparison*
String	toLowerCase()	*this string, with lowercase letters*
String	toUpperCase()	*this string, with uppercase letters*
String	replace(String a, String b)	*this string, with* `a`*s replaced by* `b`*s*
String	trim()	*this string, with leading and trailing whitespace removed*
boolean	matches(String regexp)	*is this string matched by the regular expression?*
String[]	split(String delimiter)	*strings between occurrences of* `delimiter`
boolean	equals(Object t)	*is this string's value the same as* `t`*'s?*
int	hashCode()	*an integer hash code*

Generic Programming

Java in a nutshell

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear.

Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code.

Decision-making statements (if-then, if-then-else, switch)

Looping statements (for, while, do-while)

Branching statements (break, continue, return)

Using Strings in switch Statements

In Java SE 7 and later, you can use a String object in the switch statement's expression.

Generic Programming

Java in a nutshell

An example

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
    String typeOfDay;
    switch (dayOfWeekArg) {
        case "Monday":
            typeOfDay = "Start of work week";
            break;
        case "Tuesday":
        case "Wednesday":
        case "Thursday":
            typeOfDay = "Midweek";
            break;
        case "Friday":
            typeOfDay = "End of work week";
            break;
        case "Saturday":
        case "Sunday":
            typeOfDay = "Weekend";
            break;
        default:
            throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
    }
    return typeOfDay;
}
```

Generic Programming

Java in a nutshell

Classes and Objects

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Modifiers**: A class can be public or have default access .
- **class keyword:** class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

Generic Programming

Java in a nutshell

Constructors are used for initializing new objects.
Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Object

Object is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which, interact by invoking methods.
An object consists of :

- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Objects correspond to things found in the real world.

Declaring Objects or instantiating a class

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

The **new** operator instantiates a class by allocating memory for a new object and returning a reference to that memory.

Generic Programming

Java in a nutshell

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called **fields**.
- Variables in a method or block of code—these are called **local variables**.
- Variables in method declarations—these are called **parameters**.

Field declarations are composed of three components, in order:

- Zero or more modifiers, such as public or private.
- The field's type.
- The field's name.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field.

public modifier—the field is accessible from all classes.
private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be directly accessed from the class in which they are declared. We still need access to these values, however. This can be done indirectly by adding public methods that obtain the field values for us:

Generic Programming

Java in a nutshell

Types

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

Variable Names

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions.

Some rules

the first letter of a class name would be capitalized, and
the first (or only) word in a method name would be a verb.

Generic Programming

Java in a nutshell

Method declarations have six components, in order:

1. **Modifiers**—such as public, private, and others.
2. **The return type**—the data type of the value returned by the method, or void if the method does not return a value.
3. **The method name**—the rules for field names apply to method names as well, but the convention is a little different.
4. **The parameter list in parenthesis**—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. **An exception list**—.
6. **The method body**, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Generic Programming

Java in a nutshell

Method overloading

Overloaded methods are differentiated by the number and the type of the arguments passed into the method.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Generic Programming

Java in a nutshell

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

What happens if you don’t provide any constructor?

You must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does.

If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

Generic Programming

Java in a nutshell

Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor.

The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers and reference data types, such as objects and arrays.

What if the name of the parameter is the same as one of the fields of the class?
A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to shadow the field.

Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field.

Generic Programming

Java in a nutshell

```
public class PairOfDice {
        public int die1; // Number showing on the first die.
        public int die2; // Number showing on the second die.
        public PairOfDice(int val1, int val2) {
            // Constructor. Creates a pair of dice that
            // are initially showing the values val1 and val2.
            die1 = val1; // Assign specified values
            die2 = val2; // to the instance variables.
        }
        public void roll() {
            // Roll the dice by setting each of the dice to be
            // a random number between 1 and 6.
            die1 = (int)(Math.random()*6) + 1;
            die2 = (int)(Math.random()*6) + 1;
        }
} // end class PairOfDice
```

Generic Programming

Java in a nutshell

The Garbage Collector

The Java platform allows you to create as many objects as you want, and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

Generic Programming

Java in a nutshell

Using the "**this**" Keyword

Within an instance method or a constructor, **this** is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using **this**.

Using **this** with a Field

The most common reason for using the "**this**" keyword is because a field is shadowed by a method or constructor parameter.

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Generic Programming

Java in a nutshell

Using **this** with a Constructor

From within a constructor, you can also use the **this** keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Generic Programming

Java in a nutshell

Class Methods

The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, **without the need for creating an instance of the class**, as in

```
ClassName.methodName(args)
```

Note: You can also refer to static methods with an object reference like

```
instanceName.methodName(args)
```

but this is discouraged because it does not make it clear that they are class methods.

Generic Programming

Java in a nutshell

Java Inheritance Basics

When a class inherits from a superclass, it inherits parts of the superclass methods and fields. The subclass can also override (redefine) the inherited methods. Fields cannot be overridden, but can be "shadowed" in subclasses.

Constructors are not inherited by subclasses, but a subclass constructor must call a constructor in the superclass.

Java Only Supports Singular Inheritance

Generic Programming

Java in a nutshell

Declaring Inheritance in Java

In Java inheritance is declared using the **extends** keyword. You declare that one class extends another class by using the extends keyword in the class definition.

Example

```
public class Vehicle {
    protected String licensePlate = null;

    public void setLicensePlate(String license) {
        this.licensePlate = license;
    }
}
public class Car extends Vehicle {
    int numberOfSeats = 0;

    public String getNumberOfSeats() {
        return this.numberOfSeats;
    }
}
```

Overriding Methods

In a subclass, you can override (redefine) methods defined in the superclass.

To override a method the method signature in the subclass must be the same as in the superclass. That means that the method definition in the subclass must have exactly the same name and the same number and type of parameters, and the parameters must be listed in the exact same sequence as in the superclass. Otherwise the method in the subclass will be considered a separate method.

Calling Superclass Methods

If you override a method in a subclass, but still need to call the method defined in the superclass, you can do so using the **super** reference, like this:

```
public class Car extends Vehicle {

    public void setLicensePlate(String license) {
        super.setLicensePlate(license);
    }
}
```

You can call superclass implementations from any method in a subclass, like above. It does not have to be from the overridden method itself. For instance, you could also have called super.setLicensePlate() from a method in the Car class called updateLicensePlate() which does not override the setLicensePlate() method.

Nested Classes

In Java nested classes are classes that are defined inside another class.

The purpose of a nested class is to clearly group the nested class with its surrounding class, signaling that these two classes are to be used together. Or perhaps that the nested class is only to be used from inside its enclosing (owning) class.

Java developers often refer to nested classes as inner classes, but inner classes (non-static nested classes) are only one out of several different types of nested classes in Java.

In Java nested classes are considered members of their enclosing class. Thus, a nested class can be declared public, package (no access modifier), protected and private (see access modifiers for more info). Therefore nested classes in Java can also be inherited by subclasses.

Anonymous Classes

Anonymous classes in Java are nested classes without a class name. They are typically declared as either subclasses of an existing class, or as implementations of some interface.

Anonymous classes are defined when they are instantiated.

```
public class SuperClass {

  public void doIt() {
    System.out.println("SuperClass doIt()");
  }

}
```

Generic Programming

Java in a nutshell

```
SuperClass instance = new SuperClass() {

    public void doIt() {
        System.out.println("Anonymous class doIt()");
    }
};

instance.doIt();
```

Running this Java code would result in “Anonymous class doIt()” being printed to System.out.

The anonymous class subclasses (extends) SuperClass and overrides the doIt() method.

Generic Programming
Java in a nutshell

You can declare fields and methods inside an anonymous class, but you cannot declare a constructor. You can declare a static initializer for the anonymous class instead, though. Here is an example:

```
final String textToPrint = "Text...";

MyInterface instance = new MyInterface() {

    private String text;

    //static initializer
    {  this.text = textToPrint;  }

    public void doIt() {
        System.out.println(this.text);
    }
};

instance.doIt();
```

The same shadowing rules apply to anonymous classes as to inner classes.

Java Abstract classes

A Java abstract class is a class which cannot be instantiated, meaning you cannot create new instances of an abstract class. The purpose of an abstract class is to function as a base for subclasses.

Declaring an Abstract Class in Java

In Java you declare that a class is abstract by adding the abstract keyword to the class declaration. Here is a Java abstract class example:

```
public abstract class MyAbstractClass {

}
```

The following Java code is no longer valid:

```
MyAbstractClass myClassInstance =
    new MyAbstractClass();  //not valid
```

If you try to compile the code above, the Java compiler will generate an error, saying that you cannot instantiate MyAbstractClass because it is an abstract class.

Generic Programming
Java in a nutshell

Abstract Methods
An abstract class can have abstract methods. You declare a method abstract by adding the abstract keyword in front of the method declaration.

```
public abstract class MyAbstractClass {

    public abstract void abstractMethod();
}
```

An abstract method has no implementation. It just has a method signature.

If a class has an abstract method, the whole class must be declared abstract. Not all methods in an abstract class have to be abstract methods. An abstract class can have a mixture of abstract and non-abstract methods.

In Java abstract classes are intended to be extended to create a full implementation. Thus, it is fully possible to extend an abstract class. The Java inheritance rules are the same for abstract classes as for non-abstract classes.

Generic Programming

Java in a nutshell

Interfaces

A Java interface is a bit like a Java class, except a Java interface can only contain method signatures and fields.

A Java interface is not intended to contain implementations of the methods, only the signature (name, parameters and exceptions) of the method.

However, it is possible to provide default implementations of a method in a Java interface, to make the implementation of the interface easier for classes implementing the interface.

You can use interfaces in Java as a way to achieve polymorphism.

Generic Programming
Java in a nutshell

Java Interface Example

```
public interface MyInterface {

    public String hello = "Hello";

    public void sayHello();
}
```

Just like with classes, a Java interface can be declared public or package scope (no access modifier).

The interface example above contains one variable and one method. The variable can be accessed directly from the interface, like this:

```
System.out.println(MyInterface.hello);
```

Accessing a variable from an interface is very similar to accessing a static variable in a class.

The method, however, needs to be implemented by some class before you can access it.

Generic Programming
Java in a nutshell

Implementing an Interface

Before you can really use an interface, you must implement that interface in some Java class. Here is a class that implements the MyInterface interface shown above:

```
public class MyInterfaceImpl implements MyInterface {

    public void sayHello() {
        System.out.println(MyInterface.hello);
    }
}
```

Notice the **implements** MyInterface part of the above class declaration. This signals to the Java compiler that the MyInterfaceImpl class implements the MyInterface interface.

A class that implements an interface must implement all the methods declared in the interface. The methods must have the same signature (name + parameters) as declared in the interface.

Generic Programming

Java in a nutshell

Interface Instances

Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface.

You cannot create instances of a Java interface by itself. You must always create an instance of some class that implements the interface, and reference that instance as an instance of the interface.

A Java class can implement multiple interfaces.

Generic Programming

Java in a nutshell

Lambda expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression.

Functional Interface

An interface which has only one abstract method is called functional interface.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression consists of three components.

1) Argument-list: It can be empty also.

2) Arrow-token: It is used to link arguments-list and the body of expression.

3) Body: It contains expressions and statements for lambda expression.

No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
```

One Parameter Syntax

```
(p1) -> {
//Body of single parameter lambda
}
```

Two Parameter Syntax

```
(p1,p2) -> {
//Body of multiple parameter lambda
```

Lambda Expression Example: No Parameter

```
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
    }
}
```

Generic Programming

Java in a nutshell

Java Lambda Expression Example: Single Parameter

```
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say(“Mysore"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say(“Mysore"));
    }
}
```

Java Lambda Expression Example: Multiple Parameters

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Generic Programming

Introduction

The idea—that the same code can be used for a variety of data types—is called generic programming. Parameterized types make it possible for the same class to work with many different kinds of data.

Generic programming refers to writing code that will work for many types of data.

Java 5.0 introduced parameterized types, which made it possible to create generic data structures that can be type-checked at compile time rather than at run time.

For example, if list is of type ArrayList<String>, then the compiler will only allow objects of type String to be added to list. Furthermore, the return type of list.get(i) is String, so type-casting is not necessary.

Java's parameterized classes are similar to template classes in C++

Generic Programming

Introduction

The ArrayList class is a resizable array, which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).

Elements can be added and removed from an ArrayList whenever you want.

Generic Programming

Introduction

How to use ArrayList.

Create an ArrayList object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
   System.out.println(cars);
  }
}
```

Generic Programming

Introduction

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

Example
cars.get(0);

And, there are other methods to change/ remove an item.
There is a method to find out how many elements are in the Arraylist (size)

Generic Programming

Introduction

Other Types

Elements in an ArrayList are actually objects.

In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type).

To use other types, such as int, you must specify an equivalent wrapper class: Integer.

For other primitive types, use:
Boolean for boolean, Character for char, Double for double, etc:

Generic Programming

Introduction

Create an ArrayList to store numbers (add elements of type Integer):

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

Generic Programming

Introduction

In general, we can have ArrayList<T> for any object type T.

ArrayList is just one class, but the source code works for many different types.

This is generic programming

Generic Programming

Introduction

If you have variables typeless and parameters to subroutines typeless in a programming language, does it help in generic programming?

Generic Programming

Introduction

Generic Programming in Java

Java's generic programming features have gone through several stages of development.

Early versions of Java did not have parameterized types, but they did have classes to represent common data structures. Those classes were designed to work with **Objects**; that is, they could hold objects of any type, and there was no way to restrict the types of objects that could be stored in a given data structure.

For example, ArrayList was not originally a parameterized type, so that any ArrayList could hold any type of object. This means that if list was an ArrayList, then list.get(i) would return a value of type **Object**. If the programmer was actually using the list to store Strings, the value returned by list.get(i) would have to be type-cast to treat it as a string:

```
String item = (String)list.get(i);
```

Generic Programming

Introduction

This is still a kind of generic programming, since one class can work for any kind of object, but it was closer in spirit to Smalltalk than it was to C++, since there is no way to do type checks at compile time.

Unfortunately, as in Smalltalk, the result is a category of errors that show up only at run time, rather than at compile time.

If a programmer assumes that all the items in a data structure are strings and tries to process those items as strings, a run-time error will occur if other types of data have inadvertently been added to the data structure.

In Java, the error will most likely occur when the program retrieves an **Object** from the data structure and tries to type-cast it to type String.

If the object is not actually of type String, the illegal type-cast will throw an error of type ClassCastException.

Generic Programming

Introduction

Java 5.0 introduced parameterized types, which made it possible to create generic data structures that can be type-checked at compile time rather than at run time.

For example, if list is of type ArrayList<String>, then the compiler will only allow objects of type String to be added to list.

Furthermore, the return type of list.get(i) is String, so type-casting is not necessary.

Java's parameterized classes are similar to template classes in C++ (although the implementation is very different), and their introduction moves Java's generic programming model closer to C++ and farther from Smalltalk.

It is still legal to use a parameterized class as a non-parameterized type, such as a plain ArrayList. In that case, any type of object can be stored in the data structure.

Generic Programming

Introduction

Types of Java Generics

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like ‘int’,’char’ or ‘double’.

Generic Programming

Advantages

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

Stronger type checks at compile time.

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.

Generic Programming

Evolution

Java's generic programming features have gone through several stages of development.

Early versions of Java did not have parameterized types, but they did have classes to represent common data structures.

Those classes were designed to work with Objects; that is, they could hold objects of any type, and there was no way to restrict the types of objects that could be stored in a given data structure.

ArrayList was not originally a parameterized type, so that any ArrayList could hold any type of object. This means that if list was an ArrayList, then list.get(i) would return a value of type **Object**.

If the programmer was actually using the list to store Strings, the value returned by list.get(i) would have to be type-cast to treat it as a string:
String item = (String)list.get(i);

Generic Programming

Basics

Let us write our own *type-safe* resizable array (similar to an ArrayList).
We will start off with a non-type-safe non-generic version, explain generics, and write the type-safe generic version.

Non-Type-Safe Non-Generic MyArrayList
MyArrayList - a linear data structure, similar to array, but resizable.

For the MyArrayList to hold all types of objects, we use an **Object**[] to store the elements. Since **Object** is the single root class in Java, all Java objects can be upcast to **Object** and stored in the **Object**[].

The two source files are:

Source1

Source2

Generic Programming

Basics

This MyArrayList is not *type-safe*. It suffers from the following drawbacks:

1. The upcasting to java.lang.Object is done implicitly by the compiler. But, the programmer has to explicitly downcast the Object retrieved back to their original class (e.g., String).
2. The compiler is not able to check whether the downcasting is valid at *compile-time*. Incorrect downcasting will show up only at *runtime*, as a ClassCastException. This is known as *dynamic binding* or *late binding*.
   For example, if you accidentally added an Integer object into the above list which is intended to hold String, the error will show up only when you try to downcast the Integer back to String - at runtime.

Generic Programming

Basics

Why not let the compiler do the upcasting/downcasting and check for casting error, instead of leaving it to the runtime, which could be too late? Can we make the compiler to catch this error to ensure *type safety* at runtime?

Generics Classes with Parameterized Types

JDK 5 introduced the so-called *generics* to resolve this problem.
Generics allow us to *abstract over types*.
The class designer can design a class with a *generic type*.
The users can create specialized instance of the class by providing the *specific type* during instantiation.
Generics allow us to *pass type information*, in the form of <type>, to the compiler, so that the compiler can perform all the necessary type-check during compilation to ensure type-safety at runtime.

Generic Programming

Basics

Let's take a look at the declaration of interface java.util.List<E>:

```
public interface List<E> extends Collection<E> {
   abstract boolean add(E element)
   abstract void add(int index, E element)
   abstract E get(int index)
   abstract E set(int index, E element)
   abstract E remove(int index)
   boolean addAll(Collection<? extends E> c)
   boolean containsAll(Collection<?> c)
   ......
}
```

The <E> is called the formal "type" parameter for passing type information into the generic class. During instantiation, the formal type parameters are replaced by the actual type parameters.

Generic Programming

Basics

The mechanism is similar to method invocation.
In a method's definition, we declare the formal parameters for passing data into the method.
During the method invocation, the formal parameters are substituted by the actual arguments.

For example,

```
// Defining a method
public static int max(int a, int b) {  // int a, int b are formal parameters
   return (a > b) ? a : b;
}

// Invoke the method: formal parameters substituted by actual parameters
int max1 = max(55, 66);  // 55 and 66 are actual parameters
int x = 77, y = 88;
int max2 = max(x, y);    // x and y are actual parameters
```

Generic Programming

Basics

Formal type parameters used in the class declaration have the same purpose as the formal parameters used in the method declaration.
A class can use *formal type parameters* to receive type information when an instance is created for that class.
The actual types used during instantiation are called *actual type parameters*.

Methods pass parameters through round bracket ( ), type parameters are passed through angle bracket < >.

Let's return to the List<E>. In an actual instantiation, such as a List<String>, all occurrences of the formal type parameter E are replaced by the actual type parameter String. With this additional type information, compiler is able to perform type check during compile-time and ensure that there won't be type-casting errors at runtime.

Look at the following program:
Source3

Generic Programming

Basics

Generic Type vs. Parameterized Type

A *generic type* is a type with *formal type parameter*s (e.g. List<E>); whereas a *parameterized type* is an instantiation of a generic type with *actual type arguments* (e.g., List<String>).

Formal Type Parameter Naming Convention

Use an uppercase single-character for formal type parameter. For example,

<E> for an element of a collection;

<T> for type;

<K, V> for key and value.

<N> for number

S,U,V, etc. for 2nd, 3rd, 4th type parameters

A Generic Class GenericBox

A class called GenericBox, takes a generic type parameter E, holds a content of type E. The constructor, getter and setter work on the parameterized type E. The toString() reveals the actual type of the content.

The source file is here:
Source4

The following test program creates GenericBoxes with various types (String, Integer and Double). Take note that JDK 5 introduced auto-boxing and unboxing to convert between primitives and wrapper objects.

Source5

(JDK 7) Improved Type Inference for Generic Instance Creation with the Diamond Operator <>

Before JDK 7, to create an instance of the above GenericBox, you needed to specify the type in the constructor:
GenericBox<String> box1 = new GenericBox<String>("hello");

JDK 7 introduced the *type inference* to shorten the code, as follows:
GenericBox<String> box1 = new GenericBox<>("hello");
// type inferred from the variable

Generic Programming

Type erasure

From the previous example, it seems that compiler substituted the parameterized type E with the actual type (such as String, Integer) during instantiation. If this is the case, the compiler would need to create a new class for each actual type (similar to C++'s template).

In fact, the compiler replaces all reference to parameterized type E with java.lang.Object. For example, the above GenericBox is compiled as follows, which is compatible with the code without generics:

```
public class GenericBox {
  private Object content;  // private variable
  public GenericBox(Object content) {  // Constructor
    this.content = content;
  }
  public Object getContent() {  // getter
    return content;
  }
  public void setContent(Object content) {  // setter
    this.content = content;
  }
  public String toString() {  // describe itself
    return "GenericBox[content=" + content + "(" + content.getClass() + ")]";
  }
}
```

Generic Programming

Type erasure …

The compiler performs the type checking and inserts the required downcast operator when the methods are invoked:

```
// Constructor: public GenericBox(E content)
GenericBox<String> box1 = new GenericBox<>("hello");
// Knowing E=String, compiler performs the type check

// Getter: public E getContent()
String str = (String)box1.getContent();
// Compiler inserts the downcast operator to downcast Object to String
```

In this way, the same class definition is used for all the types.
Most importantly, the bytecodes are compatible with those without generics.
This process is called *type erasure*.

Generic Programming

Type erasure …

For example, GenericBox<Integer> and GenericBox<String> are compiled into the same runtime class GenericBox.

```
public class GenericBoxTypeTest {
   public static void main(String[] args) {
      GenericBox<Integer> box1 = new GenericBox<>(123);
      GenericBox<String> box2 = new GenericBox<>("hello");
      System.out.println(box1.getClass() == box2.getClass());  //true
(same runtime class)
      System.out.println(box1.getClass());  //class GenericBox
      System.out.println(box2.getClass());  //class GenericBox
   }
}
```

Demo of the above program.

Generic Programming

Type-Safe MyGenericArrayList<E>

Let us rewrite the MyArrayList example using generics:

The program is here.

Dissecting the Program

MyGenericArrayList<E> declare a generics class with a *formal type parameter* <E>.

During an actual invocation, e.g., MyGenericArrayList<String>, a specific type <String>, or *actual type parameter*, replaced the formal type parameter <E>.

Generic Programming

Type erasure …

Type Erasure

Behind the scene, generics are implemented by the Java compiler as a front-end conversion called *erasure*, which translates or rewrites code that uses generics into non-generic code to ensure backward compatibility.

This conversion erases all generic type information. The formal type parameter, such as <E>, are replaced by Object by default (or by the upper bound of the type). When the resulting code is not type correct, the compiler inserts a type casting operator.

Hence, the translated code is as shown in this file:

When the class is instantiated with an actual type parameter, e.g. MyGenericArrayList<String>, the compiler performs type check to ensures add(E e) operates on only String type. It also inserts the proper downcasting operator to match the return type E of get().

Generic Programming

Type erasure …

With generics, the compiler is able to perform type checking during compilation to ensure type safety at runtime.

Unlike "template" in C++, which creates a new type for each specific parameterized type, in Java, a generics class is only compiled once, and there is only one single class file which is used to create instances for all the specific types.

Generic Programming

Generic methods

Generic Methods
Other than generic class already described, we can also define methods with generic types.

For example, the java.lang.String class, which is non-generic, contains a generic method .transform() defined as follows:

```
// Class java.lang.String
public <R> R transform(Function<? super String, ? extends R> f)
// JDK 12
```

A generic method should declare formal type parameters, which did not appear in the class statement, (e.g. <R>) ***preceding the return type***.

The formal type parameters can then be used as *placeholders* for return type, method's parameters and local variables within a generic method, for proper type-checking by compiler.
An example is here.

Generic Programming

Generic methods

In this example, we define a static generic method Array2List() to append an array of generic type E to a List<E>. In the method definition, we need to declare the generic type <E> before the return-type void.
Similar to generic class, when the compiler translates a generic method, it replaces the formal type parameters using *erasure*. All the generic types are replaced with type Object by default (or the upper bound of type). The translated version is as follows:

```
public static void Array2List(Object[] arr, List lst) {
   for (Object e : arr) lst.add(e);
}
```

When the method is invoked, the compiler performs type check and inserts downcasting operator during retrieval.
Generics have an optional syntax for specifying the type for a generic method. You can place the actual type in angle brackets <>, between the dot operator and method name. For example,
GenericMethodTest.<Integer>Array2List(arr, lst);
The syntax makes the code more readable and also gives you control over the generic type in situations where the type might not be obvious.

Generic Programming

Basics

Constructors

In a generic class, type parameters appear in the header that declares the class, but not in the constructor:

```
class Pair<T, U> {
        private final T first;
        private final U second;
        public Pair(T first, U second) { this.first=first; this.second=second; }
        public T getFirst() { return first; }
        public U getSecond() { return second; }
}
```

The type parameters T and U are declared at the beginning of the class, not in the constructor. However, actual type parameters are passed to the constructor whenever it is invoked:

```
Pair<String, Integer> pair = new Pair<String, Integer>("one",2);
assert pair.getFirst().equals("one") && pair.getSecond() == 2;
```

So, we need not have

public Pair <T, U> (T first, U second) { this.first=first; this.second=second; }

Generic Programming

Generic subtypes

Generic Subtypes

Knowing that String is a subtype of Object. Consider the following lines of codes:

```
// String is a subtype of Object
Object obj = "hello";
// A supertype reference holding a subtype instance
System.out.println(obj);  //hello

// But ArrayList<String> is not a subtype of ArrayList<Object>
ArrayList<Object> lst = new ArrayList<String>();
//compilation error: incompatible types: ArrayList<String> cannot be converted to ArrayList<Object>.
```

When we try to upcast ArrayList<String> to ArrayList<Object>, it triggers a compilation error "incompatible types". This is because ArrayList<String> is NOT a subtype of ArrayList<Object>, even through String is a subtype of Object.

Generic Programming

Generic subtypes

This error is against our intuition on inheritance.
Why? Consider these two statements:
List<String> strLst = new ArrayList<>(); // 1
List<Object> objLst = strLst; // 2

//compilation error: incompatible types:
List<String> cannot be converted to List<Object>

Line 2 generates a compilation error.

But if line 2 succeeds and some arbitrary objects are added into objLst, strLst will get "corrupted" and no longer contains only Strings, as references objLst and strLst share the same value.

Hence, List<String> is NOT a subtype of List<Object>, although String is a subtype of Object.

Generic Programming

Generic subtypes

On the other hand, the following is valid:
// ArrayList is a subtype of List
List<String> lst = new ArrayList<>(); // valid

That is, ArrayList<String> is a subtype of List<String>, since ArrayList is a subtype of List and both have the same parametric type String.

In summary:

1. Different instantiation of the same generic type **for different concrete type arguments** (such as List<String>, List<Integer>, List<Object>) have NO type relationship.
2. Instantiations of super-sub generic types for the **same actual type argument exhibit the same super-sub type relationship**, e.g., ArrayList<String> is a subtype of List<String>.

Generic Programming

Array subtype

Array Subtype?

String[] is a subtype of Object[]. But if you upcast a String[] to Object[], you cannot re-assign value of non-String type. For example:

```
import java.util.Arrays;
public class ArraySubtypeTest {
   public static void main(String[] args) {
      String[] strArr = {"apple", "orange"};
      Object[] objArr = strArr;  // upcast String[] to Object[]
      System.out.println(Arrays.toString(objArr));
      objArr[0] = 123;  // compile ok, runtime error
      //Exception in thread "main" java.lang.ArrayStoreException:
java.lang.Integer
   }
}
```

Why is this?

Arrays carry runtime type information about their component type. Hence, you CANNOT use E[] in your generic class, but need to use Object[], as in the MyGenericArrayList<E>.

Generic Programming

Generic subtypes

Wildcards <? extends T>, <? super T> and <?>

Suppose we want to write a *generic method called* printList(List<.>) to print the elements of a List.

If we define the method as printList(List<Object> lst), then it can only accept an argument of List<object>, but not List<String> or List<Integer>.

For example, look at the following program;

Program

Unbounded Wildcard <?>

To resolve this problem, a wildcard (?) is provided in generics, which stands for *any unknown type*. For example, we can rewrite our printList() as follows to accept a List of any unknown type.

public static void printList(List<?> lst) { for (Object o : lst)
System.out.println(o); }

The unbounded wildcard <?> is, at times, too relaxed in type.

Generic Programming

Generic subtypes

Upper Bounded Wildcard <? extends T>
To write a generic method that works on List<Number> and the subtypes of Number, such as List<Integer>, List<Double>, we could use an upper bounded wildcard <? extends Number>.

In general, the wildcard <? extends T> stands for type T and T's subtypes.

An example

List<? extends Number> accepts List of Number and any subtypes of Number, e.g., List<Integer> and List<Double>.

Another example,
//List<Number> lst = new ArrayList<Integer>();
//compilation error: incompatible types: ArrayList<Integer> cannot be converted to List<Number>

List<? extends Number> lst = new ArrayList<Integer>(); // valid

Generic Programming

Generic subtypes

Revisit Unbounded Wildcard <?>

Clearly, <?> can be interpreted as <? extends Object>, which accepts ALL Java classes.

You should use <?> only if:

1. The implementation depends only on methods that are provided in the Object class.
2. The implementation does not depend on the type parameter.

Generic Programming

Generic subtypes

Lower Bounded Wildcard <? super T>

The wildcard <? super T> matches type T, as well as T's supertypes. In other words, it specifies the lower bound type.

Suppose that we want to write a generic method that puts an Integer into a List. To maximize flexibility, we also like the method to work on List<Integer>, as well as List<Number>, List<Object> that can hold Integer.

In this case, we could use the less restrictive lower bounded wildcard <? super Integer>, instead of simply List<Integer>.

An example

Generic Programming

Generic subtypes

Bounded Type Parameters

Wildcard types don't solve all of our problems.

They allow us to **generalize method definitions** so that they can work with collections of objects of various types, rather than just a single type.

However, they do not allow us to restrict the types that are allowed as type parameters in a generic class or method definition. Bounded types exist for this purpose.

Generic Programming

Generic subtypes

Upper Bounded Type Parameters <T extends TypeName>
A bounded parameter type is a **generic type that specifies a bound for the generic**, in the form of <T extends TypeName>, e.g., <T extends Number> accepts Number and its subclasses (such as Integer and Double).

For example, the static method add() takes a type parameter <T extends Number>, which accepts Number and its subclasses (such as Integer and Double).

Generic Programming

Generic subtypes

In general, a bounded type parameter “T extends **SomeType**” means roughly “a type, T, that is either equal to SomeType or is a subclass of **SomeType**; the upshot is that any object of type T is also of type **SomeType**, and any operation that is defined for objects of type **SomeType** is defined for objects of type T.

The type **SomeType** doesn’t have to be the name of a class. It can be any name that represents an actual object type. For example, it can be an interface or even a parameterized type.

Generic Programming

Generic subtypes

Bounded types and wildcard types are clearly related. They are, however, used in very different ways.

A bounded type can be used only as a formal type parameter in the definition of a generic method, class, or interface. A wildcard type is used most often to declare the type of a formal parameter in a method and cannot be used as a formal type parameter.

One other difference, by the way, is that, in contrast to wildcard types, bounded type parameters can only use “extends”, never “super”.

Generic Programming

Generic subtypes

How does the compiler treat the bounded generics?

As mentioned, by default, all the generic types are replaced with type Object during the code translation. However, in the case of <T extends Number>, the generic type is replaced by the type Number, which serves as the *upper bound* of the generic types.

```
public class UpperBoundedTypeParamMaximumTest {
   public static <T extends Comparable<T>> T maximum(T x, T y) {
      // Need to restrict T to Comparable and its subtype for .compareTo()
      return (x.compareTo(y) > 0) ? x : y;
   }

   public static void main(String[] args) {
      System.out.println(maximum(55, 66));    // 66
      System.out.println(maximum(6.6, 5.5));  // 6.6
      System.out.println(maximum("Monday", "Tuesday"));  // Tuesday
   }
}
```

Generic Programming

Generic subtypes

By default, Object is the *upper-bound* of the parameterized type.
<T extends Comparable<T>> changes the upper bound to the Comparable interface, which declares an abstract method compareTo() for comparing two objects.
The compiler translates the above generic method to the following codes:

```
public static Comparable maximum(Comparable x, Comparable y)
{ // replace T by upper bound type Comparable
// Compiler checks x, y are of the type Comparable
// Compiler inserts a type-cast for the return value
return (x.compareTo(y) > 0) ? x : y; }
```

When this method is invoked, e.g. via maximum(55, 66), the primitive ints are auto-boxed to Integer objects, which are then implicitly upcast to Comparable. The compiler checks the type to ensure type-safety. The compiler also inserts an explicit downcast operator for the return type.

Generic Programming

Generic subtypes

That is,
(Comparable)maximum(55, 66);
(Comparable)maximum(6.6, 5.5);
(Comparable)maximum("Monday", "Tuesday");

We do not have to pass an actual type argument to a generic method. The compiler infers the type argument automatically, based on the type of the actual argument passed into the method.

Note:
<T extends Comparable<T>>
Note that the type parameter T is also part of the signature of the super interface Comparable<T>. and how does the above piece of code help achieve mutual comparability?
It ensures that you can only compare objects of type T. Without the type bound, Comparable compares any two Objects. With the type bound, the compiler can ensure that only two objects of type T are compared.

Generic Programming

Generic subtypes

Bounded Type Parameter for Generic Class

The bounded type parameter <T extends ClassName> can also be applied to generic class, e.g.,

An example

Generic Programming

Generic Collection Classes – an overview

The Java platform includes a *collections framework*. A *collection* is an object that represents a group of objects. A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
The primary advantages of a collections framework are that it:
Reduces programming effort by providing data structures and algorithms so you don't have to write them yourself.
Increases performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
Reduces the effort required to learn APIs by requiring you to learn multiple ad hoc collection APIs.
Reduces the effort required to design and implement APIs by not requiring you to produce ad hoc collections APIs.
Fosters software reuse by providing a standard interface for collections and algorithms with which to manipulate them.

Generic Programming

Collection framework

The collections framework consists of:

Collection interfaces. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.

General-purpose implementations. Primary implementations of the collection interfaces.

Legacy implementations. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.

Special-purpose implementations. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior.

Concurrent implementations. Implementations designed for highly concurrent use.

Wrapper implementations. Add functionality, such as synchronization, to other implementations.

Generic Programming

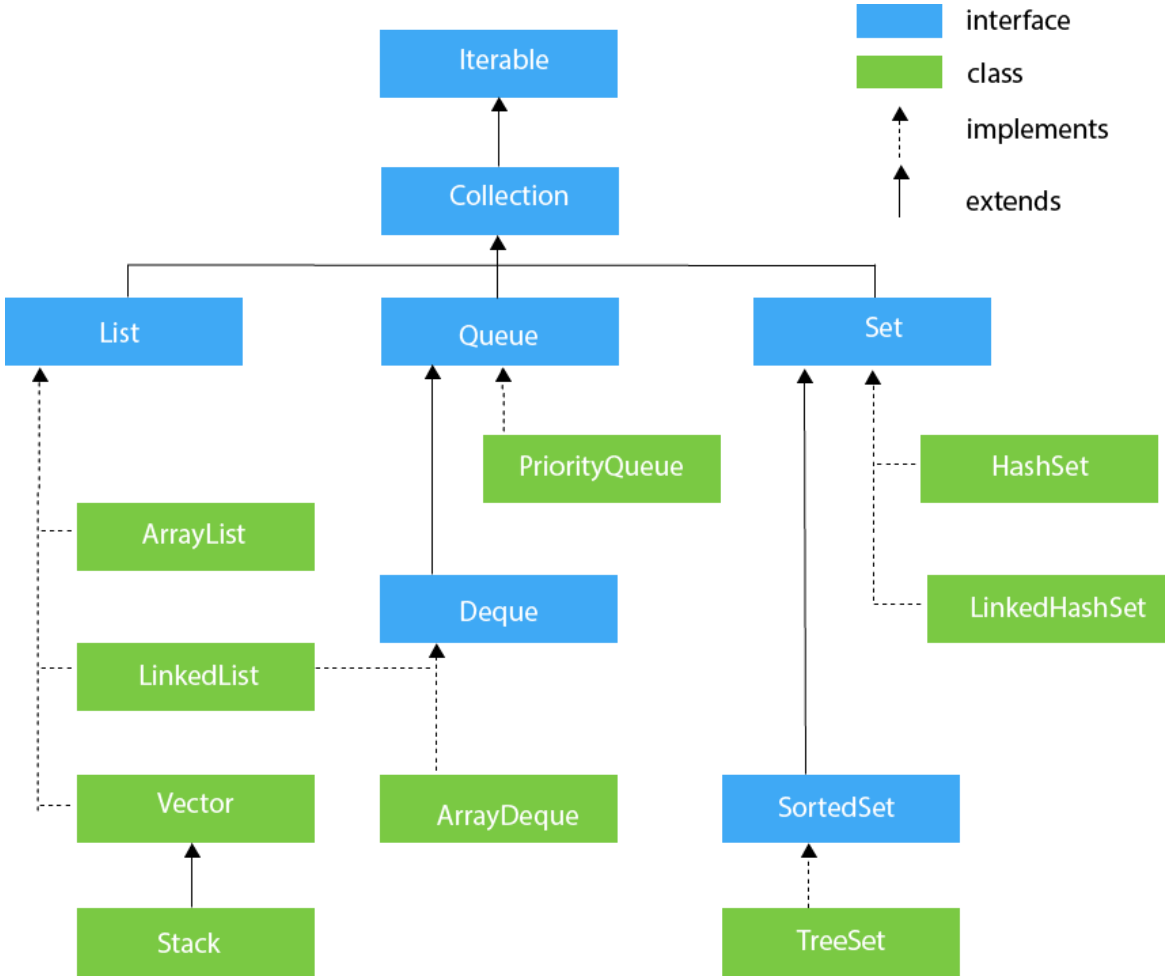
Collection framework

Convenience implementations. High-performance "mini-implementations" of the collection interfaces.
Abstract implementations. Partial implementations of the collection interfaces to facilitate custom implementations.
Algorithms. Static methods that perform useful functions on collections, such as sorting a list.
Infrastructure. Interfaces that provide essential support for the collection interfaces.
Array Utilities. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure.

Generic Programming

Collection framework - Hierarchy

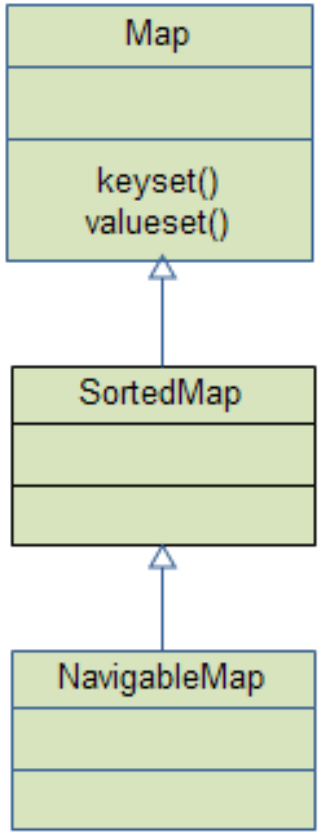
There are two "groups" of interfaces: Collections and Maps.



Generic Programming

Collection framework - Hierarchy

Map interface hierarchy



Generic Programming
Collection framework

Java Collection

The Java Collection interface represents the operations possible on a generic collection, like on a List, Set, Stack, Queue and Deque. For instance, methods to access the elements based on their index are available in the Java Collection interface.

Java Set

The Java Set interface represents an unordered collection of objects. Unlike the List, a Set does not allow you to access the elements of a Set in any guaranteed order. There are Set implementations that order elements based on their natural ordering, but the Set interface itself provides no such guarantees.

Java SortedSet

The Java SortedSet interface represents an ordered collection of objects. Thus, the elements in the SortedSet can be iterated in the sorted order.

Java NavigableSet

The Java NavigableSet interface is an extension of the SortedSet interface with additional methods for easy navigation of the elements in the NavigableSet.

Generic Programming
Collection framework

Java Map

The Java Map interface represents a mapping between sets of keys and values. Both keys and values are objects. You insert a key + value pair into a Map, and later you can retrieve the value via the key - meaning you only need the key to read the value out of the Map again later.

Java SortedMap

The Java SortedMap interface is an extension of the Map interface, representing a Map where the keys in the Map are sorted. Thus, you can iterate the keys stored in the SortedMap in the sorted order, rather than the kind-of-random order you iterate them in a normal Map.

Java NavigableMap

The Java NavigableMap interface is an extension of the SortedMap interface which contains additional methods for easy navigation of the keys and entries in the NavigableMap.

Java Stack

The Java Stack class represents a classical stack data structure, where elements can be pushed to the top of the stack and popped off from the top of the stack again later

Generic Programming
Collection framework

Java Queue

The Java Queue interface represents a classical queue data structure, where objects are inserted into one end of the queue and taken off the queue in the other end of the queue. This is the opposite of how you use a stack.

Java Deque

The Java Deque interface represents a double ended queue, meaning a data structure where you can insert and remove elements from both ends of the queue.

Java Iterator

The Java Iterator interface represents a component that is capable of iterating a Java collection of some kind. For instance, a List or a Set. You can obtain an Iterator instance from the Java Set, List, Map etc.

Java Iterable

The Java Iterable interface is very similar in responsibility to the Java Iterator interface. The Iterable interface allows a Java Collection to be iterated using the for-each loop in Java.

Generic Programming

Collection framework

The **java.util** package contains all the classes and interfaces for the Collection framework.

Generic Programming
Collection framework

Some of the methods of “Collection” interface

Method	Description
public boolean add(E e)	used to insert an element in this collection.
public boolean addAll(Collection<? extends E> c)	used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	used to delete an element from the collection
public boolean removeAll(Collection<?> c)	used to delete all the elements of the specified collection from the invoking collection.
public boolean retainAll(Collection<?> c)	used to delete all the elements of invoking collection except the specified collection.
public int size()	returns the total number of elements in the collection.

Generic Programming
Collection framework

Some of the methods of "Collection" interface

Method	Description
public boolean contains(Object element)	used to search an element.
public boolean containsAll(Collection<?> c)	used to search the specified collection in the collection.
public Iterator iterator()	returns an iterator.
public Object[] toArray()	converts collection into array.
public <T> T[] toArray(T[] a)	converts collection into array. Here, the runtime type of the returned array is that of the specified array.
public boolean isEmpty()	checks if collection is empty.

Methods of “Iterator” interface

Method	Description
public boolean hasNext()	returns true if the iterator has more elements otherwise it returns false.
public Object next()	returns the element and moves the cursor pointer to the next element.
public void remove()	removes the last elements returned by the iterator. This method is rarely used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,
Iterator<T> iterator()
It returns the iterator over the elements of type T

Generic Programming
Collection framework

Our own generic programs

Generic1

Generic2

Generic3

Some sample programs using the collection framework:

Example1

Example2

Example3

The working of C# programs

C# is a general-purpose, strongly typed, lexically scoped, functional, object-oriented, and component-oriented programming language.

When you compile a C# program, the source code is converted to an intermediate code which is known as **Common Intermediate Language (CIL)** or **Intermediate Language Code (ILC or IL code)**. This CIL or IL Code can run on any operating system because C# is a *Platform Independent* Language.

After converting the C# source code to Common Intermediate Language (CIL) or Intermediate Language Code (ILC or IL code, the intermediate code needs to be converted to machine understandable code.

Generic Programming

C# - Introduction

C# uses the *.NET Framework* and as part of this .NET Framework, the *Virtual Machine component* manages the execution of programs written in any language that uses the . NET Framework.

This virtual machine component is known as **Common Language Runtime (CLR)** which translates the CIL or IL code to native code or machine understandable code or machine instructions.

This process is called the **Just-In-Time (JIT) Compilation or Dynamic Compilation** which is the way of compiling code during the execution of a program at run time only.

Generic Programming

C# - Introduction

Once the C# programs are compiled, they're physically packaged into **Assemblies**. An assembly is a file that contains one or more namespaces and classes.

As the number of classes and namespaces in program grows, it is physically separated by related namespaces into separate assemblies.

Assemblies typically have the file extension **.exe** or **.dll,** depending on whether they implement applications or libraries respectively, where EXE stands for *Executable* and DLL stands for *Dynamic Link Library*.

An EXE (Executable) file represents a program that can be executed and a DLL (Dynamic Link Library) file includes code (Eg: Library) that can be reused across different programs.

Generic Programming

The first C# program

```
// C# program to print Hello World!
using System;

// namespace declaration
namespace HelloWorldApp {

        // Class declaration
        class Pesu {

                // Main Method
                static void Main(string[] args) {

                        // statement printing Hello World!
                        Console.WriteLine("Hello World!");

                        // To prevent the screen from running and closing quickly
                        Console.ReadKey();
                }
        }
}
```

Generic Programming
C# programming

A quick explanation

using System: System is a namespace which contains the commonly used types. It is specified with a **using** System directive.

namespace HelloWorldApp: Here namespace is the keyword which is used to define the namespace. *HelloWorldApp* is the user-defined name given to namespace.

class Pesu: Here *class* is the keyword which is used for the declaration of classes. *Pesu* is the user-defined name of the class.

static void Main(string[] args): Here *static* keyword tells us that this method is accessible without instantiating the class. *void* keyword tells that this method will not return anything. *Main()* method is the entry point of our application. In our program, *Main()* method specifies its behavior with the statement *Console.WriteLine("Hello World!");*.

Console.WriteLine(): Here *WriteLine()* is a method of the Console class defined in the System namespace.

Console.ReadKey(): This is for the VS.NET Users. This makes the program wait for a key press and prevents the screen from running and closing quickly.

Save the program with an extension .cs.

Compile the program using the compiler csc.exe

This would create an executable Pesu.exe which can be executed.

Generic Programming

Java and C# comparison

Keywords

Java	C#	Java	C#
abstract	abstract	native	extern
new	new		
break	break	null	null
case	case	package	namespace
catch	catch	private	private
class	class	protected	internal
const	const	public	public
continue	continue	return	return
default	default	static	static
do	do		
else	else	super	base
enum	enum	switch	switch
extends	:	synchronized	lock

Generic Programming
Java and C# comparison

Keywords

Java	C#	Java	C#
false	false	this	this
final	sealed	throw	throw
finally	finally	throws	n/a
for	for / foreach		
goto	goto	true	true
if	if	try	try
implements	:	… (varargs)	params
import	using	void	void
instanceof	is	volatile	volatile
interface	interface	while	while

Note:
The const and goto keywords in Java have no function.
The C# const and goto keywords are operational.

Generic Programming

Java and C# comparison

Operators

Java	C#	Description
x.y	x.y	Member access “dot” operator
f(x)	f(x)	Method invocation operator
a[x]	a[x]	Array element access operator
++, --	++, --	Increment and decrement operators (pre and postfix)
new	new	Object instantiation operator
instanceof	is	Type verification operator
(T)x	(T)x	Explicit cast operator
+, -	+, -	Addition and subtraction operators (binary). Positive and negative operators (unary)
+	+	String concatenation operator
!	!	Logical negation operator
&&, ||	&&, ||	Conditional AND and OR operators

Generic Programming

Java and C# comparison

Operators

Java	C#	Description
~	~	Bitwise complement operator
&, |, ^	&, |, ^	Bitwise AND, OR, and XOR operators
<<, >>	n/a	Signed left-shift and right-shift operators
>>> >>		Unsigned right-shift operator
*, /, %	*, /, %	Multiply, divide, and modulus operators
==, !=	==, !=	Is-equal-to and is-not-equal-to operators
<, >, <=,>=	<, >, <=,>=	Relational less-than, greater-than, less-than-or-equalto, and greater-than-or-equal-to operators
x?y:z	x?y:z	Conditional operator
=	=	Assignment operator

Generic Programming

Java and C# comparison

Common data types

Java	C#
byte	sbyte
char	char
double	double
float	float
int	int
long	long
short	short
String	string
Object	object
Date	DateTime

Generic Programming
Java and C# comparison

C# keywords not found in Java, or with different behavior

base	Provides access to members in a parent class
operator	Declares an overloaded operator
checked	Enables arithmetic overflow checking
out	Declares an output parameter (method call). Declares a covariant type parameter(generic interface)
delegate	Declares a type-safe reference to a method
override	Declares a method that overrides a method in a base class
event	Declares an event
protected	Declares a member that is accessible only within the class and descendant classes (different semantics from Java protected keyword)
explicit	Declares a narrowing conversion operator
readonly	Declares a field to be readonly

Generic Programming

Java and C# comparison

C# keywords not found in Java, or with different behavior

implicit	Declares a widening conversion operator
ref	Declares a reference to a value type
in	Declares a contravariant type parameter for a generic interface
struct	Defines a new value type
new	Declares a method that hides a method in a base class (method modifier)
virtual	Declares a member that can be overridden

Generic Programming
Java and C# comparison

Java wrapper classes mapped to C# structs

Java Wrapper	C# Struct
Boolean	Boolean
Byte	Byte
Character	Char
n/a	Decimal
Double	Double
Float	Float
Integer	Int32
Long	Int64
Short	Int16
Void	n/a

Generic Programming
Java and C# comparison

Program and Class example

C# program

Java Program

C# Struct and Enum example

Example2

Example3

Generic Programming
Java and C# comparison

Inheritance in C#
Example5

Interface example

Interface implementation in C#
Sample3

Interface implementation in Java
JavaSample3

Using switch in C#
Sample4

Generic Programming
Java and C# comparison

The delegate is a reference type data type that defines the method signature. You can define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

Using Delegate
Example6

Using properties in C#
A property in C# is a member of a class which is used to set and get the data from a data field of a class.
A property in C# is never used to store data, it just acts as an interface to transfer the data.

We use the Properties as they are the public data members of a class, but they are actually special methods called **accessors**.
Example7

Generic Programming
Java and C# comparison

Reflection

Reflection is a mechanism in programming to implement generic code that can work for all types of objects. It helps in recognizing the format for the objects at runtime and invoking the methods of that object and accessing the fields of these objects. Using these descriptors, one can easily invoke instance methods and access their variables.

Reflection in C#

Reflection is the process of describing the metadata of types, methods and fields in a code. The namespace **System.Reflection** enables you to obtain data about the loaded assemblies, the elements within them like classes, methods and value types.

Sample program is here.

Generic Programming

Generics

Generics is a new feature in C# 2.0. Generics allow you to define type-safe data structures without specifying an actual data type. Generics are similar to C++ templates in concept, but are very different in implementation and capabilities.

To see what problems Generics solve, consider a very simple data structure such as the stack, providing the basic Push() and Pop() operations.

Typically, you would like each stack instance to store instances of various types. For example, you may need a stack of ints, a stack of strings and so on.

In C# 1.0 you have to use an object-based stack, meaning that the internal data type used in the stack is an instance of object and methods must interact with object instances:

Generic Programming

C# program

```
// Sample implementation
public class Stack
{
    object[] m_altems = null;

    public void Push( object item )
    { ... }

    public object Pop()
    { ... }
}

// Sample Usage
Stack s = new Stack();
s.Push( 1 );
s.Push( 2 );
int n = (int)s.Pop();
```

Generic Programming

Generics

There are two main problems in this approach:

Performance. When using value types, they have to be boxed in order to push and store them as objects (a reference type), and they have to be unboxed when popping them off the stack. Boxing and unboxing incur a significant overhead as boxing allocates a new object on the heap which increases the pressure on the managed heap resulting in more garbage collections, which obviously affects performance. Even when using reference types, there is still a performance hit as they have to be casted up to an object when pushing and down to the original type when popping.

Type-safety. This is a more severe problem because the compiler allows you to cast any type to and from **object**. You therefore lose compile-time type-safety. For example, the following code compiles fine (when it should not) and raises an InvalidCastException at runtime:

Generic Programming

C# program

```
Stack stack = new Stack();
stack.Push( 1 );
string s = (string)stack.Pop();    // InvalidCastException. We should
cast to an int.
```

In C# 1.0, you can overcome these two problems by providing a type-specific (and hence type-safe) stack. For example, you would have to implement an IntStack for integers and StringStack for strings:

```
// Sample implementation
public class IntStack
{
   int[] m_altems = null;

   public void Push( int item )
   { ... }

   public int Pop()
   { ... }
}
```

Generic Programming

C# program

```
// Sample implementation
public class StringStack
{
    string[] m_altems = null;

    public void Push( string item )
    { ... }

    public string Pop()
    { ... }
}
```

And so on. Obviously, while this approach solves the performance and type-safety problems, it does introduce a third and more serious problem - productivity impact. Writing type-safe data structures is a tedious, repetitive and error-prone task.

Enter Generics.

Generic Programming

Generics

What Are Generics?

Generics allows you to define type-safe classes without compromising on type-safety, performance, or productivity. The main approach is: you implement a generic data structure only once, and then instantiate it with a specific data type. Classes, interfaces, structures, delegates, instance and static methods can all be generic types. Here is how you define and use a generic Stack:

```
public class Stack<T>
{
    T[] m_aItems = null;

    public void Push(T item)
    { /* ... */ }

    public T Pop()
    { /* ... */ }
}
```

Generic Programming

Generics

```
Stack<int> stack = new Stack<int>();
stack.Push( 10 );
stack.Push( 20 );
int n1 = stack.Pop();
int n2 = stack.Pop();
```

Generics have native support in IL and the CLR.
When you compile generics C# code, the compiler converts it to IL just like any other compiled code, however, the IL will contain placeholders for the actual specific types.

In addition, the metadata for the generic code contains generic information.

When a program provides a specific type instead of the generic type parameter T, the specific type will replace the generic type parameter T as if generics were never involved.

Generic Programming

Generics

Generics should be used whenever a data structure or a utility class can be expressed in a generic version, and not the object-based version.

Typically, collections and data structures such as linked lists, queues, binary trees etc will offer generic support, but generics are not limited to data structures.

Applying Generics

Note on naming Generics conventions which are used throughout:

T is the generic type parameter. In other words, a generic type parameter is a placeholder for types.

Stack<T> is the generic type.

int in Stack<int> is the type argument. In other words, the type argument is the type the client specifies to use instead of the generic type parameter.

Because of the native support for generics in IL and the CLR, most CLR-compliant languages can take advantage of generic types. The following simple examples show how to apply generics:

Generic Programming

Generics

A simple demo program is here.

Another program which uses a generic class with a generic constructor, generic member variable and a generic method is shown below.

G_Demo2

Generic Programming

Generics

```
// Example 2 - This class represents a node whose types are generic
internal class Node<K, I>
{
    public K            key;
    public I            item;
    public Node<K, I>   nextNode;

    public Node()
    {
        key      = default(K);       // Note use of default
        item     = default(I);       // Note use of default
        nextNode = null;
    }

    public Node(K k, I i, Node<K, I> next)
    {
        key      = k;
        item     = i;
        nextNode = next;
    }
}
```

Generic Programming

Generics

```
// The following class represents a linked list whose node types are generic.
// The fact the LinkedList uses the same names as the node for the
// generic type arguments is purely for readability purposes. Other names
// for K and I could have been used.

internal class LinkedList<K, I>
{
    internal Node<K, I> m_head;

    public LinkedList()
    {
        // We must provide type arguments when instantiating a generic type.
        // we can use the linked list's own generic type arguments
        m_head = new Node<K, I>();
    }
```

```
public void AddHead(K k, I i)
    {
        // Again, we must provide type arguments when instantiating a generic
        // type. We can use the linked list's own generic type arguments
        Node<K, I> node = new Node<K, I>(k, i, m_head.nextNode);
        m_head.nextNode = node;
    }
}

LinkedList<int, string> list = new LinkedList<int, string>();
list.AddHead(10, "Ten");
list.AddHead(20, "Twenty");
```

```
LinkedList<double, DateTime> list2 =
                                    new LinkedList<double, DateTime>();
list2.AddHead(10.0, DateTime.Now.AddDays(1));
list2.AddHead(20.0, DateTime.Now.AddDays(2));

using List = LinkedList<int, string>;
// Providing an alias for a particular combination of types.
// The scope of aliasing is the scope of the file
List list = new List();
list.AddHead( 10, "hello" );
```

Generic Programming

Generics

Note the following naming conventions for generic types (assume the generic type is LinkedList<K,I>):

LinkedList<int,string> is a closed constructed type, or simply a constructed type.

LinkedList<int, I>, LinkedLinst<K, string> and LinkedList<K,I> are all called open constructed types.

Generics

Generic Constraints
Generic code can contain code that is incompatible with the specified type argument.

Generic constraints impose certain constraints or limitations on the type arguments in order to ensure that the generic code behaves correctly.

For example, a generic function inside a generic class may need to call ICloneable.Clone on the type argument, meaning that the generic function assumes that the type argument implements ICloneable. If the supplied type argument did not implement ICloneable, a runtime exception will be generated and the code fails.

Generic constraints solve these kinds of problems.

Generic Programming

Generics

There are three types of generic constraints:

Derivation Constraint:
This indicates to the complier that the generic type parameter derives from a base type such as an interface or a base class.

Default Constructor Constraint:
This indicates to the complier that the generic type parameter has a default constructor (a public constructor with no arguments.)

Reference/value type Constraint:
This indicates to the complier that the generic type parameter is a reference or a value type.

Generic Programming

Generics

It is important to note that while constraints are optional, they can be essential when developing a generic type.

Without them, the compiler takes a more conservative and type-safe approach and only allows access to object-level functionality in your generic type parameters.

Also note that constraints are part of the generic type metadata so that the compiler can take advantage of them.

The compiler reads the constraints from the metadata and only allows type arguments that comply with the constraints, thus enforcing type safety.

Derivation Constraint

This constraint uses the **where** keyword to indicate to the compiler that a generic type parameter derives from some base type. The compiler will generate an error (CS0309) if a type argument did not derive from the given base class. The following examples illustrate:

Example 1: Basic derivation constraint.

```
public interface I
{
    /* ... */
}

public class A : I
{
    /* ... */
}
```

Generic Programming

Generics

```
public class B
{
    /* ... */
}

public class C<T> where T : I    // Generic type parameter T must derive
from interface I
{
    /* ... */
}

C<A> c1 = new C<A>();    // A derives from interface  I, hence code
compiles.
C<B> c2 = new C<B>();    // B does not derive from interface I, hence
compiler error CS0309
```

Generic Programming

Generics

Example 2: All derivation constraints must appear after the actual derivation list of the generic class.

```
public class C<T> : IEnumerable<T> where T : I
// C<T> derives from IEnumerable<T>.
// Also generic type argument T must derive from interface I
{
    /* ... */
}
```

Example 3: You can provide a derivation constraint for each generic type argument.

```
public class A<T, K> where T : MyBase
// A base class in a derivation constraint list cannot be sealed or
// static. And there can be only one base class.
                    where K : IComparable<K>
{ /* ... */ }
```

Generic Programming

Generics

Example 4: You can constrain a base class and/or multiple interfaces on the same generic type argument. If a base class is specified, it must appear first in the derivation constraint list.

```
public class A<T,K> where T: MyBase, IEnumerable, ICloneable<T>
{ /* ... */ }
```

Example 5: When dealing with a derivation constraint, the type argument can specify the base type itself, and not necessarily a derived class.

```
public interface IMyInterface
{ /* ... */ }

public class MyClass : IMyInterface
{ /* ... */ }

public class A<T> where T : IMyInterface
{ /* ... */ }

A<IMyInterface> a = new A<IMyInterface>();
```

Generic Programming

Constraints

Default Constructor Constraint

This constraint uses the **new** keyword to indicate to the compiler that a generic type parameter supports a public default constructor (a constructor with no parameters).

For example, suppose you want to instantiate inside the generic class a new object from the type argument. The compiler will generate error CS0304 if the type argument did not support a default constructor. The following example illustrates:

```
// Does not compile
public class D<T>
{
    public T m_nID = new T();   // error CS0304: Cannot create an instance of the variable type 'T' because it does not have the new() constraint

    // ...
}
```

Generic Programming

Constraints

```
// Same example as above, but uses the new constraint
public class D<T> where T : new()
{
    public T m_nID = new T();    // OK
    // ...
}

// Same example as above but combining the default constructor constraint
with a derivation constraint (default constructor constraint must appear last)
public class D<T> where T : SomeBase, new()
{
    /* ... */
}
```

Generic Programming

Generics

Reference/Value type Constraint

This constraint uses the struct/class keywords to indicate to the compiler that a generic type parameter is a value type/reference type, respectively. The following example illustrates:

```
public class A<T> where T : struct        // T must be a value type
{ /* ... */ }

public class B<T> where T : class         // T must be a reference type
{ /* ... */ }

A<int> a1    = new A<int>();         // OK. Type argument is value-type
B<string> b1 = new B<string>();      // OK. Type argument is reference-type
A<string> a2 = new A<string>();      // error CS0453: The type 'string' must be a
non-nullable value type in order
                                     // to use it as parameter 'T' in the generic type or method
'Generics.A<T>'
```

Note that reference/value constraint cannot be used with a base class constraint, but can be combined with any other constraint. When used, a reference/value constraint must appear first.

Generic Programming

Generics

Generics and Casting
Note the following points:

C# compiler allows you to implicitly cast generic type parameters to **object** or to constraint-specified types. Such casting is type-safe because any incompatibility is discovered at compile-time:

```
internal interface IMyInterface
{
    void foo();
}

internal class MyBaseClass
{
    public void bar() { /* ... */ }
}
```

Generic Programming

Generics

```
internal class MyClass<T> where T : MyBaseClass, IMyInterface
{
    public void SomeMethod(T t)
    {
        // Can implicitly cast to a constraint-specified interface
        IMyInterface itf = t;

        // Can implicitly cast to a constraint-specified class
        MyBaseClass c = t;

        // Can implicitly cast to object
        object o = t;
    }
}
```

Generic Programming

Generics

However, explicit casting is required when casting from a generic type parameter to an interface not in the constraint-list. You cannot explicitly cast a generic type parameter to a class not in the constraint-list - you have to use a temporary object:

```
internal interface IMyInterface
{
    void foo();
}

internal class MyBaseClass
{
    public void bar() { /* ... */ }
}
```

Generic Programming

Generics

```
internal class MyClass<T>           // Note this class has no derivation-
constraints
{
    public void SomeMethod(T t)
    {
        IMyInterface itf = t;       // error CS0266: Cannot implicitly convert type
                                    // 'T' to 'Generics.Casting2.IMyInterface'.
                                    // An explicit conversion exists
        MyBaseClass mbc = t;        // error CS0029: Cannot implicitly convert
                                    // type 'T' to 'Generics.Casting2.MyBaseClass'

        IMyInterface itf2 = (IMyInterface)t;            // OK
        MyBaseClass mbc2  = ((MyBaseClass)(object)t);   // OK
    }
}
```

Generic Programming

Generics

Explicit casting is dangerous as it may throw an exception at run time if the type argument does not derive from the type you wish to cast to.

A better approach is to use the **is** and **as** operators:

```
internal interface IMyInterface
{
    void foo();
}

internal class MyBaseClass
{
    public void bar() { /* ... */ }
}
```

Generic Programming

Generics

```
internal class MyClass<T>
{
    public void SomeMethod(T t)
    {
        // Using 'is'
        IMyInterface itf = null;
        if ( t is IMyInterface)
            itf = (IMyInterface)t;

        // Using 'as‘
        // 'as' keyword checks the compatibility of one object type with another
          // object type. In case of compatible, it will return the value of the new
          // object type otherwise, null will be returned.
        MyBaseClass mbc = t as MyBaseClass;
        if (mbc != null)
            mbc.bar();
    }
}
```

Generic Programming

Generics

Generics and Inheritance
Note the following points:

When deriving from a generic class, a type argument must be provided instead of the generic type parameter:

```
internal class MyGenericBaseClass<T>            { /* ... */ }
internal class MyClass : MyGenericBaseClass<int> { /* ... */ }
```

When deriving from a generic class, if the sub-class is generic too, you can use the subclass generic type parameter for the generic base class:

```
internal class MyGenericBaseClass<T>                    { /* ... */ }
internal class MyGenericSubClass<T> : MyGenericBaseClass<T>  { /* ... */
}
```

Generic Programming

Generics

When deriving from a generic class, if the sub-class is generic too and the base class specifies constraints, you must repeat any base-class constraints:

```
internal class MyGenericBaseClass2<T> where T : ICloneable
{ /* ... */ }
internal class MyGenericSubClass2<T> : MyGenericBaseClass2<T> where
T : ICloneable { /* ... */ }
```

Generic Programming

Generics

When deriving from a generic class, a type argument must be provided instead of the generic type parameter. And if the base generic class has a virtual method whose signature uses generic type parameters, the overridden method in the derived class must provide the corresponding types in the method signature:

```
internal class MyGenericBaseClass3<T>
{
    public virtual void SomeMethod(T t) { /* ... */ }
}

internal class MyClass2 : MyGenericBaseClass3<int>
{
    public override void SomeMethod(int n)
    {
        { /* ... */}
    }
}
```

Generic Programming

Generics

You can define generic interfaces, generic base classes, and generic abstract methods:

```
internal interface IMyInterface<T>
{
    T SomeMethod(T t);
}

internal abstract class MyAbstractGenericClass<T>
{
    public abstract void SomeMethod(T t);
}

internal class MyDerivedGenericClass<T> : MyAbstractGenericClass<T>
{
    public override void SomeMethod(T t) { /* ... */ }
}
```

Generic Programming

Generics

You cannot use + or += operators on generic type parameters. The recommended solution is to compensate using interfaces (recommended solution) or abstract classes:

```
public class Calculator<T>
{
    public T Add(T lhs, T rhs)
    {
        return lhs + rhs;          // error CS0019: Operator '+' cannot be applied to
                                   // operands of type 'T' and 'T'
    }
}

public interface ICalculator<T>
{
    T Add(T lhs, T rhs);
}
```

Generic Programming

Generics

```
public class Calculator2 : ICalculator<int>
{
    public int Add(int lhs, int rhs)
    {
        return lhs + rhs;
    }
}
```

Generic Programming

Generics

A generic method is a method that is declared with type parameters:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
        T temp; temp = lhs;
        lhs = rhs;
        rhs = temp;
}
```

One way to call the method by using int for the type argument:

```
public static void TestSwap() {
        int a = 1; int b = 2;
        Swap<int>(ref a, ref b);
        System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to Swap is equivalent to the previous call:

```
Swap(ref a, ref b);
```

Generic Programming

Generics

The same rules for type inference apply to static methods and instance methods.

The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value.

Therefore type inference does not work with methods that have no parameters.

Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures.

The compiler applies type inference logic to all generic methods that share the same name.

In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Generic Programming

Generics

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
class SampleClass<T>
{
        void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning **CS0693** because within the method scope, the argument supplied for the inner T hides the argument supplied for the outer T.

If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in GenericList2<T> in the following example.

Generic Programming

Generics

```
class GenericList<T>
{
        // CS0693 void SampleMethod<T>() { }
}

class GenericList2<T>
{       //No warning
        void SampleMethod<U>() { }
}
```

Use constraints to enable more specialized operations on type parameters in methods. This version of Swap<T>, now named SwapIfGreater<T>, can only be used with type arguments that implement IComparable<T>.

Generic Programming

Generics

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

```
void DoWork() { }

void DoWork<T>() { }

void DoWork<T, U>() { }
```

Generic Programming

Generics

// When calling a method that defines its own generic type parameter, you can either provide the type, or you can let the compiler infer the type based on the type of the parameter that was passed in (this compiler ability is called 'generic type inference‘).

```
// Both calls below to Method2 compile
MyClass<int> obMyClass = new MyClass<int>();
obMyClass.Method2<string>("hello", 20);        // OK
obMyClass.Method2("inferred", 30);             // OK

// Calling a method that defines constraints on its own generic type parameters

obMyClass.Method3<double>(10.0);
```

Generic Programming

Generics

Another example relating to instance generic methods but for subclasses as well:

```
internal class MyClass2        // Not a generic class!
{
    // You can define generic methods even if the class does not use generics
    // at all
    public virtual void Method1<T>(T t) where T : struct
    {
        Trace.WriteLine(" t = " + t);
    }
}
```

Generic Programming

Generics

```
internal class MySubClass2 : MyClass2
{
    // Sub-class implementation must override the base virtual generic method
but should not re-specify  any constraints at all, else error CS0460 is
generated: 'Constraints for override and explicit interface implementation
methods are inherited from the base method, so they cannot be specified
    // directly'
    public override void Method1<T>(T t)
                                    //No constraints can be specified here
    {
        Trace.WriteLine(" t = " + t);

        // If calling the base class implementation, you can either specify the type
        // argument or rely on type inference
        base.Method1<T>(t);         // OK
        base.Method1(t);            // OK
    }
}
```

Generic Programming

Generics

Static Methods

```
internal class MyClass<T>
{
    // You can define static methods that use generic type parameters
    public static T SomeMethod(T t)
    {
        Trace.WriteLine("t = " + t);
        return t;
    }

    // Just like instance methods, static methods can define method-specific
    // generic type parameters and constraints
    public static void SomeMethod2<X>(X x, T t) where X : class
    {
        Trace.WriteLine("x = " + x + " t = " + t);
    }
}
```

Generic Programming

Generics

```
// Calling a static generic method
int n = Generics.StaticMethods.MyClass<int>.SomeMethod(10);

// Calling a static generic method that defines method-specific generic type
// parameters. Just like instance methods, you can either explicitly specify
// the type argument or rely on type inference

MyClass<double>.SomeMethod2<string>("hello", 12.34);  // OK

MyClass<double>.SomeMethod2("hello", 12.34);          // OK

MyClass<double>.SomeMethod2<int>(12, 12.34);
// error CS0452: The type 'int' must be a reference type in order to use
// it as parameter 'X' in the generic type or method
'MyClass<T>.SomeMethod2<X>(X, T)'
```

Generic Programming

Generics

A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
Del<int> m2 = Notify;
```

Generic Programming

Generics

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
class Stack<T> {
        T[] items;
        int index;
        public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
private static void DoWork(float[] items) { }
        public static void TestStack() {
                Stack<float> s = new Stack<float>();
                Stack<float>.StackDelegate d = DoWork;
}
```

Generic Programming

Generics

Generic delegates are especially useful when it comes to events. You can literally define a limited set of generic delegates, distinguished only by the number of the generic type parameters they require, and use these delegates for all of your event handling needs. The following code illustrates the use of a generic delegate and a generic event-handling method:

```
 // Generic event handling
internal delegate void GenericEventHandler<S, A>(S sender, A args) where A :
System.EventArgs;    // Note constraints

internal class Publisher
{
    public event GenericEventHandler<Publisher, System.EventArgs> MyEvent;
    public void FireEvent()
    {
        MyEvent( this, new EventArgs());
    }
}
```

Generic Programming

Generics

Generics and Reflection

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for non-generic types.

In .NET Framework 2.0, several new members were added to the type class to enable run-time information for generic types.

The System.Reflection.Emit namespace also contains new members that support generics.

Generic Programming

Generics

System.Type member name	Description
isGenericType	Returns true if a type is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
GetGenericTypeDefinition	Returns the underlying generic type definition for the current constructed type.
GetGenericParameterConstraints	Returns an array of Type objects that represent the constraints on the current generic type parameter.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.

Generic Programming

Generics

System.Type member name	Description
GenericParameterAttributes	Gets a combination of GenericParameterAttributes flags that describe the special constraints of the current generic type parameter.
GenericParameterPosition	For a Type object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
isGenericParameter	Gets a value that indicates whether the current Type represents a type parameter of a generic type or method definition.
isGenericTypeDefinition	Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.

Generic Programming

Generics

System.Type member name	Description
DeclaringMethod	Returns the generic method that defined the current generic type parameter, or null if the type parameter was not defined by a generic method.
MakeGenericType	Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.
ContainsGenericParameters	Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types have not been supplied.

Generic Programming

Generics

In addition, members of the **MethodInfo** class enable run-time information for generic methods.

System.Reflection.MemberInfo Member Name	Description
isGenericMethod	Returns true if a method is generic.
GetGenericArguments	Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
GetGenericMethodDefinition	Returns the underlying generic method definition for the current constructed method.
ContainsGenericParameters	Returns true if the method or any of its enclosing types contain any type parameters for which specific types have not been supplied.

Generic Programming

Generics

System.Reflection.MemberInfo Member Name	Description
isGenericMethodDefinition	Returns true if the current MethodInfo represents the definition of a generic method.
MakeGenericMethod	Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

Generic Programming

Generics vs. Interfaces

Interfaces and generics serve different purposes. Interfaces are about defining a contract between a service consumer and a service provider. As long as the consumer programs against the interface, it can use any other service provider that supports the same interface. This allows switching service providers without affecting client code.

On the other hand, generics are about defining and implementing a service without committing to the actual types used. As such, interfaces and generics are not mutually exclusive, in fact they complement each other. Good programming practices often combine interfaces and generics.

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com