Generic Programming
Unit 5

Compiled by
M S Anand

Department of Computer Science

Generic Programming

Text Book:

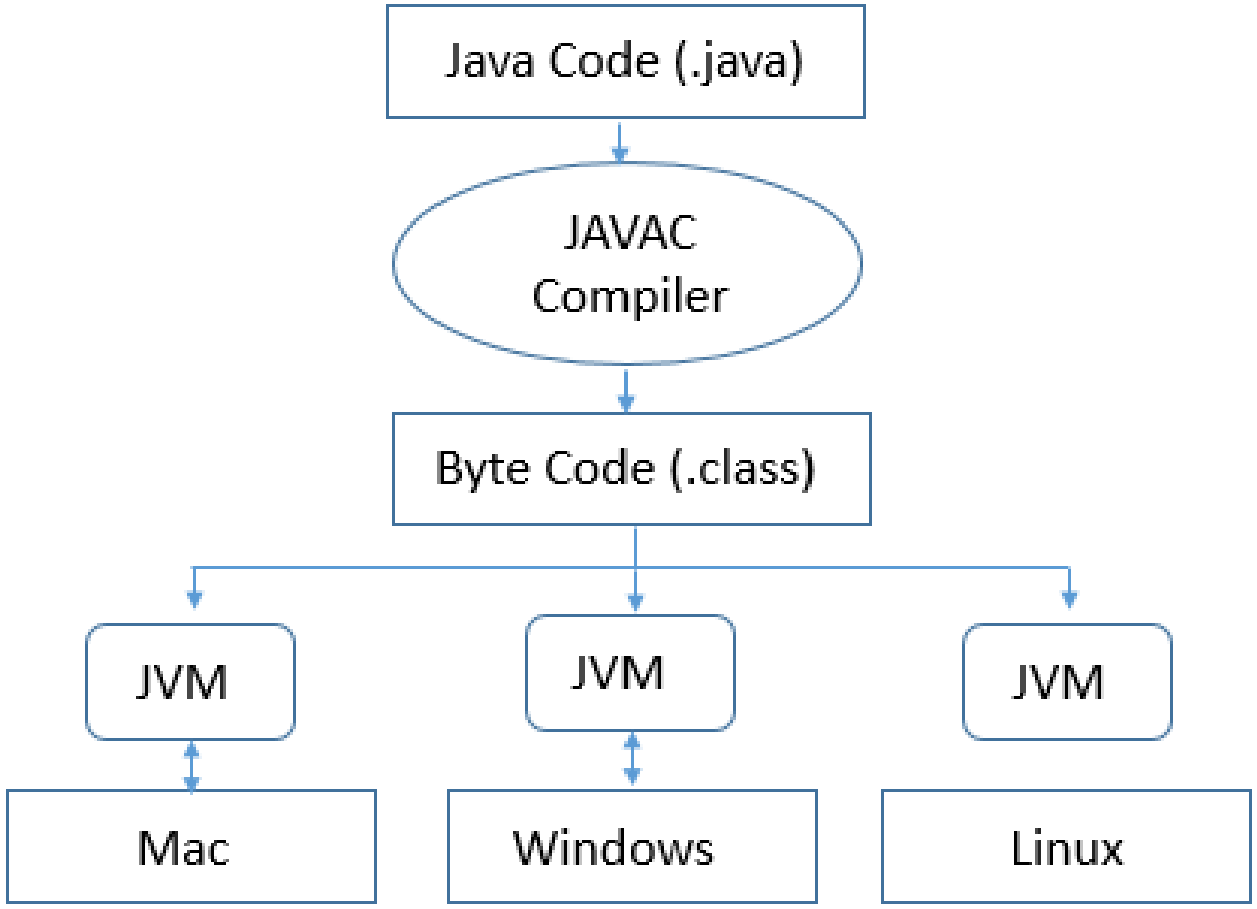
1: “STL Tutorial and Reference”, Musser, Derge and Saini, 2nd Edition, Addison-Wesley, 2001.

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.

Generic Programming

Java in a nutshell

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

Generic Programming
Java in a nutshell

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.

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));
    }
}
```

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@yahoo.com