13MCA32 – PROGRAMMING USING JAVA
II INTERNAL ANSWER KEY (ODD 2016)

1a) Define a package. Explain the creation of the package by the name shape to illustrate it. (7)

Packages are containers for classes that are used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world. This is the general form of the package statement:

package pkg; //pkg is the name of the package.

Following is the program that creates a package shape and includes 3 classes in it (circle1, square1, triangle1)

```
package shape;
public class circle1
{       public void cirdisp1()
        {
                System.out.println("formula of circle is 3.142*r*r");
}       }
```

```
package shape;
public class square1
{       public void sqrdisp1()
        {       System.out.println("formula of Square =length*width");
}       }
```

```
package shape;
public class triangle1
{       public void tridisp1()
        {       System.out.println("formula of triangle =0.5*length*breadth");
}       }
```

The following code shows how to include a package.

```
import shape.*;
public class prgm7
{       public static void main(String[] args)
        {       triangle1 tri=new triangle1();
                square1 sqr=new square1();
                circle1 cir=new circle1();
                tri.tridisp1();
                sqr.sqrdisp1();
                cir.cirdisp1();
}       }
```

1b)Explain access specifiers in java with an example.(3)

Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

• Subclasses in the same package

• Non-subclasses in the same package

• Subclasses in different packages

• Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. Table below sums up the interactions. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

2a) What is an exception in java? Explain about compile time exceptions and run time exceptions. (5)

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
```

```
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Compile time error - the java compiler can't compile the code, often because of syntax errors. Typical candidates:

- missing brackets
- missing semicolons
- access to private fields in other classes
- missing classes on the classpath (at compile time)

Runtime error - the code did compile, can be executed but crashes at some point, like you have a division by zero.

- using variable that are actually null (may cause NullPointerException)
- using illegal indexes on arrays
- accessing ressources that are currently unavailable (missing files, ...)
- missing classes on the classpath (at runtime)

2b) Write a program to illustrate NullPointerException. (5)

```
class ThrowDemo
{       static void demoproc()
        {       try
                {       throw new NullPointerException("demo");
                }
                 catch(NullPointerException e)
                {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                }
        }
        public static void main(String args[])
        {       try
                {       demoproc();
                }
                 catch(NullPointerException e)
                {       System.out.println("Recaught: " + e);
                }
        }
}
```

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throw new NullPointerException("demo");
Here, new is used to construct an instance of NullPointerException. Many of Java's builtin run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by
Throwable.

3a) What is an interface? How to implement multiple interface? (5)

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

```
interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Output:Hello
Welcome

3b) What is thread priority? Explain it with an example. (5)

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

• A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

• A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

To set a thread's priority, use the setPriority( ) method, which is a member of Thread. This is its general form:

```
final void setPriority(int level)
```

Here, level specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread. You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

```
final int getPriority( )
```

```
// Demonstrate thread priorities.
class clicker implements Runnable
{       long click = 0;
        Thread t;
        private volatile boolean running = true;
        public clicker(int p)
        {
                t = new Thread(this);
                t.setPriority(p);
        }
        public void run()
        {       while (running)
                {       click++;
                }
        }
        public void stop()
        {       running = false;
        }
        public void start()
        {       t.start();
        }
}
class HiLoPri
{       public static void main(String args[])
        {       Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
```

```
            clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
            clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
            lo.start();
            hi.start();
            try
            {       Thread.sleep(10000);
            }
            catch (InterruptedException e)
            {       System.out.println("Main thread interrupted.");
            }
            lo.stop();
            hi.stop();
            // Wait for child threads to terminate.
            try
            {       hi.t.join();
                    lo.t.join();
            }
             catch (InterruptedException e)
            {       System.out.println("InterruptedException caught");
            }
            System.out.println("Low-priority thread: " + lo.click);
            System.out.println("High-priority thread: " + hi.click);
    }
}
```

The output of this program, shown as follows when run under Windows, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got the majority of the CPU time.

Low-priority thread: 4408112
High-priority thread: 589626904

4a) Explain enumeration using an example. Explain the usage of values() and valueOf().(5)

An enumeration is a list of named constants. In Java, an enumeration defines a class type. An enumeration can have constructors, methods, and instance variables. An enumeration is created using the enum keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants. Each is implicitly declared as a public, static final member of Apple. Furthermore, their type is the type of the enumeration in which they are declared, which is Apple in this case. Thus, in the language of Java, these constants are called self-typed, in which “self” refers to the enclosing enumeration. Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an enum using new. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares ap as a variable of enumeration type Apple:

```
Apple ap;
```

Because ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns ap the value RedDel:

```
ap = Apple.RedDel;
```

Notice that the symbol RedDel is preceded by Apple.

```
// An enumeration of apple varieties.
enum Apple {
Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo {
public static void main(String args[])
{
Apple ap;
ap = Apple.RedDel;
// Output an enum value.
System.out.println("Value of ap: " + ap);
System.out.println();
ap = Apple.GoldenDel;
// Compare two enum values.
if(ap == Apple.GoldenDel)
System.out.println("ap contains GoldenDel.\n");
// Use an enum to control a switch statement.
switch(ap) {
case Jonathan:
System.out.println("Jonathan is red.");
break;
case GoldenDel:
System.out.println("Golden Delicious is yellow.");
break;
case RedDel:
System.out.println("Red Delicious is red.");
break;
case Winesap:
System.out.println("Winesap is red.");
break;
case Cortland:
System.out.println("Cortland is red.");
break;
}
}
}
```

The output from the program is shown here:

```
Value of ap: RedDel
ap contains GoldenDel.
        Golden Delicious is yellow.
```

4b) What is the importance of generics? Explain with an example. (5)

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.

```
class Gen<T>
{	T ob; // declare an object of type T
	// Pass the constructor a reference to
	// an object of type T.
	Gen(T o)
	{	ob = o;
	}
	// Return ob.
	T getob()
	{	return ob;
	}
	// Show type of T.
	void showType()
	{	System.out.println("Type of T is " +
		ob.getClass().getName());
	}
}
class GenDemo
{	public static void main(String args[])
	{	// Create a Gen reference for Integers.
		Gen<Integer> iOb;
		// Create a Gen<Integer> object and assign its
		// reference to iOb. Notice the use of autoboxing
		// to encapsulate the value 88 within an Integer object.
		iOb = new Gen<Integer>(88);
		// Show the type of data used by iOb.
		iOb.showType();
		// Get the value in iOb. Notice that
		// no cast is needed.
		int v = iOb.getob();
		System.out.println("value: " + v);
		System.out.println();
		// Create a Gen object for Strings.
		Gen<String> strOb = new Gen<String>("Generics Test");
		// Show the type of data used by strOb.
		strOb.showType();
		// Get the value of strOb. Again, notice
		// that no cast is needed.
		String str = strOb.getob();
		System.out.println("value: " + str);
	}
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

5a) Explain the way of creating our own exception sub class with an example. (5)

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. Exception defines constructors.

Exception( )

Exception(String msg)

The first form creates an exception that has no description. The second form lets you specify a description of the exception. The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception
{       private int detail;
        MyException(int a)
        {       detail = a;
        }
        public String toString()
        {       return "MyException[" + detail + "]";
        }
}
class ExceptionDemo
{       static void compute(int a) throws MyException
        {       System.out.println("Called compute(" + a + ")");
                if(a > 10)
                        throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[])
        {       try
                {       compute(1);
                        compute(20);
                }
                catch (MyException e)
                {       System.out.println("Caught " + e);
                }
        }
}
```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays thevalue of the exception. The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code. Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

5b) Explain the usage of keywords try, throw, catch, finally, throws with their syntax. (5)

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

This is the general form of an exception-handling block:

```
try
{       // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{       // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{       // exception handler for ExceptionType2
}
// ...
finally
{       // block of code to be executed after try block ends
        }
```

The program below shows how try, catch, throw, throws and finally can be used.

```
class Test
{       static void check() throws ArithmeticException
        {       System.out.println("Inside check function");
                throw new ArithmeticException("demo");
        }
        public static void main(String args[])
        {       try
                {       check();
                }
                catch(ArithmeticException e)
                {       System.out.println("caught" + e);
                }
                finally
                {       System.out.println("finally is always executed.");
                }
        }
        }
```

6a) Explain the two ways of creating threads with an example. (5)

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Implementing Runnable: The easiest way to create a thread is to create a class that implements the Runnable interface.

Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:
public void run( )
Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.
After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:
Thread(Runnable threadOb, String threadName)
In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName. After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:
void start( )
Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
```

```
}
}
```
The output produced by this program is as follows
```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread. Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
```

```
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
```

The output produced by this program is as follows

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

6b) Explain how java achieves synchronization using an example. (5)

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. You can synchronize your code in two ways.

- Using Synchronized Methods
- The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {
// statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor. Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.

class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
```

```
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
 Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. Output

```
[Hello]
[Synchronized]
[World]
```

7a) Explain Autoboxing and Autounboxing with an example. (5)

Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

```
class AutoBox2 {
// Take an Integer parameter and return
// an int value;
static int m(Integer v) {
return v ; // auto-unbox to int
}
public static void main(String args[]) {
// Pass an int to m() and assign the return value
// to an Integer. Here, the argument 100 is autoboxed
// into an Integer. The return value is also autoboxed
// into an Integer.
Integer iOb = m(100);
System.out.println(iOb);
}
}
```

This program displays the following result:

100

In the program, notice that m( ) specifies an Integer parameter and returns an int result. Inside main( ), m( ) is passed the value 100. Because m( ) is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

7b) Explain Generic methods and Generic constructors with example. (5)

Generic methods :

Methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Following example illustrates how we can print an array of different type using a single Generic method −

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }
```

```
  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array
    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array
    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

This will produce the following result −

```
Array integerArray contains:
1 2 3 4 5
Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains:
H E L L O
```

Generic constructors :

It is also possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```
// Use a generic constructor.
class GenCons {
private double val;
<T extends Number> GenCons(T arg) {
val = arg.doubleValue();
}
void showval() {
System.out.println("val: " + val);
}
}
class GenConsDemo {
public static void main(String args[]) {
GenCons test = new GenCons(100);
GenCons test2 = new GenCons(123.5F);
test.showval();
test2.showval();
}
```

}

The output is shown here:

```
val: 100.0
val: 123.5
```

Because GenCons( ) specifies a parameter of a generic type, which must be a subclass of Number, GenCons( ) can be called with any numeric type, including Integer, Float, or Double. Therefore, even though GenCons is not a generic class, its constructor is generic.