

Java New Features

- Generics
- Annotations
- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style for loop
- Variable-length arguments (varargs)
- Static import

Generics

Generics

- Concept of any collection is same irrespective of its types stored.
 - A list of names (string)
 - A set of numbers (int)
 - A list of employees (object).
- Till jdk 1.5 Collection framework provided classes whose methods accepted only parameters of type `java.lang.Object`.
- This required casting data back to respective type when extracting values from collection.
 - The process was error prone and there was a performance overhead.
- Jdk 1.5 introduced generics. It allowed collections to be created for specific type of values.
- Generics allows to abstract over types.

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections

Generics

- **Generics** means **parameterized types**.
- The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces.
- Using Generics, it is possible to create classes that work with different data types.
- An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Code 9-10 Using Non-generic Collections

```
ArrayList list = new ArrayList();  
list.add(0, new Integer(42));  
int total = ((Integer)list.get(0)).intValue();
```

Code 9-11 Using Generic Collections

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = list.get(0).intValue();
```

It is common practice to use upper case letters for type parameters. Java libraries use:

- E for element type of collection
- K and V for key-value pairs
- T for all the other types

```
Public class Stack<T> {  
    private T[] data;  
    private int topIndexLocation;  
  
    public Stack(int defaultCapacity) {  
        data = (T[]) new Object[defaultCapacity];  
        topIndexLocation = -1;  
    }  
    public boolean isEmpty() {  
        return (topIndexLocation == -1);  
    }  
    public boolean isFull() {  
        return (topIndexLocation == data.length-1);  
    }  
}
```



```
public void push(T value) {  
    if (!isFull()) {  
        topIndexLocation++;  
        data[topIndexLocation] = value;  
    } else {  
        throw new  
IndexOutOfBoundsException("Stack is full");  
    }  
}
```

```
public T pop()
{
    if (!isEmpty())
    { T valueToBeRemove = data[topIndexLocation];
      data[topIndexLocation] = null;
      topIndexLocation--;
      return valueToBeRemove; }
    else { return null; } }

public int size() { return topIndexLocation+1; }
```

```
public class Main {  
    public static void main(String[] args) {  
        Stack<Integer> intStack = new Stack<>(3);  
        Stack<String> stringStack1 = new Stack<>(6);  
        Stack<Double> doubleStack = new Stack<>(100);  
  
        .....  
  
    }  
}
```

Annotation

- A type of metadata.
- Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes.
- Annotations are used to provide supplementary information about a program.
- Annotations start with '@'.
- Annotations are meta-meta-objects which can be used to describe other meta-objects. Meta-objects are classes, fields and methods.
- An annotation needs to be interpreted in one way or another to be useful.
- Annotations can be interpreted at development-time by the IDE or the compiler, or at run-time by a framework.

Annotation

- Annotations in [Java](#) provide additional information to the compiler and [JVM](#).
- An annotation is a tag representing metadata about [classes](#), [interfaces](#), variables, [methods](#), or fields.
- Annotations do not impact the execution of the code that they annotate. Some of the characteristics of annotations are:
 - Begin with '@'
 - Do not alter the execution of the program
 - Provide supplemental information and help to link metadata with elements of a program such as classes, variables, constructs, methods, etc.
- Are different from comments since they can affect how the program is treated by the compiler

Java.lang.annotation.Annotation

Standard (Built-in)
Annotations

Custom Annotations

General Purpose Annotations
(java.lang package)


@Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface

Meta Annotations
(java.lang.annotation package)

@Inherited
@Documented
@Target
@Retention
@Repeatable

Built-in Annotations

- Annotations are metadata or *data about data*.

```
@Override  annotation  
public String toString() {  
    return "Written inside derived class";  
}
```

- Code will work even without override.
- `@Override` informs compiler to throw an error if overridden method is not there in the parent
- Prior to JDK5, this information was provided in XML files externally and processing XML outside the code was additional overhead
- But annotations are part of the code therefore easier to process

Functional Interfaces

- An interface with **exactly one abstract** method is called Functional Interface.

```
interface Car{  
    void door();  
}
```

- `@FunctionalInterface` annotation is added so that we can mark an interface as functional interface.

```
@FunctionalInterface  
interface Car{  
    void door();  
}
```


User-defined (Custom)

- To annotate program elements, i.e. variables, constructors, methods, etc. user-defined annotations can be used.
- The user-defined annotations can be applied to the elements i.e. variables, constructors, classes, methods just before their declaration.
- There are three types of annotations.
- Marker Annotation
- Single-Value Annotation
- Multi-Value Annotation

Uses

- Documentation
- Testing framework, e.g. JUnit
- WebServices
- IoC container e.g. as Spring
- Serialization, e.g. XML
- Aspect-oriented programming (AOP), e.g. Spring AOP
- Application servers, e.g. EJB container, Web Service
- Object-relational mapping (ORM), e.g. Hibernate, JPA
- and many more...

Retention Policy

- A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the `java.lang.annotation.RetentionPolicy` enumeration.
- They are `SOURCE`, `CLASS`, and `RUNTIME`.
- An annotation with a retention policy of `SOURCE` is retained only in the source file and is discarded during compilation.
- An annotation with a retention policy of `CLASS` is stored in the `.class` file during compilation. However, it is not available through the JVM during run time.
- An annotation with a retention policy of `RUNTIME` is stored in the `.class` file during compilation and is available through the JVM during run time. Thus, `RUNTIME` retention offers the greatest annotation persistence. A retention policy is specified for an annotati

Obtaining Annotations at Run Time by Use of Reflection

- Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of `RUNTIME`, then they can be queried at run time by any Java program through the use of reflection.
- Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the `java.lang.reflect` package.
- The `java.lang.Class` class provides many methods that can be used to get metadata, examine and change the run time behaviour of a class.
- The `java.lang.class` and `java.lang.reflect` packages provide classes for java reflection.

Example

- Import

```
java.lang.annotation.Annotation;import  
java.lang.annotation.Documented;import  
java.lang.annotation.ElementType;import  
java.lang.annotation.Inherited;import  
java.lang.annotation.Retention;import  
java.lang.annotation.RetentionPolicy;import  
java.lang.annotation.Target;
```

@Documented

@Inherited

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@interface SmartPhone {

String os() default "Windows";

int version() default 8;}

Using annotation

```
@SmartPhone(os="Android", version = 6)
```

```
class NokiaASeries {  
    String model;  
    int size;  
    public NokiaASeries(String model, int size) {  
        this.model = model;  
        this.size = size;  
    }  
}
```

Main Class

```
public class AnnotationDemo {  
    public static void main(String[] args) {  
        // printing normal variable values  
        NokiaASeries obj = new NokiaASeries("fire", 5);  
        System.out.println(obj.model);  
        System.out.println(obj.size);  
    }  
}
```


Reflection

```
Class c = obj.getClass();    // get class first using object
Annotation an = c.getAnnotation(SmartPhone.class);
    // get Annotation object from class
    SmartPhone s = (SmartPhone) an;
    // typecast annotation object to your annotation
    System.out.println(s.os());
    System.out.println(s.version());
}
```

Autoboxing

- Autoboxing in Java is a process of converting a ***primitive data type*** into an object of its corresponding wrapper class. For example, converting int to Integer class, long to Long class or double to Double class, etc.

Code 6-6 Examples of Primitive Autoboxing

```
int pInt = 420;  
Integer wInt = pInt; // this is called autoboxing  
int p2 = wInt; // this is called autounboxing
```

The J2SE version 5.0 compiler will now create the wrapper object automatically when assigning a primitive to a variable of the wrapper class type. The compiler will also extract the primitive value when assigning from a wrapper object to a primitive variable.

When we pass a primitive data type as a parameter to a method but that method expects an object of the wrapper class related to that primitive data type.

```
public class Example1
{
    public static void myMethod(Integer num)
    {
        System.out.println(num);
    }
    public static void main(String[] args)
    {
        //passed int (primitive type), but compiler automatically
        converts it to the Integer object.
        myMethod(5);
    }
}
```

When we work with ***collection framework*** classes, the compiler performs autoboxing in java.

```
public class Example3
{
    public static void main(String[] args)
    {
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        //Autoboxing: int primitive to Integer
        arrayList.add(11);
        arrayList.add(22);
        System.out.println(arrayList);
    }
}
```

Unboxing

- Unboxing in Java is an automatic conversion of an object of a wrapper class to the value of its respective primitive data type by the compiler.
- For example converting Integer class to int datatype, converting Double class into double data type, etc.

When we pass an object of a wrapper class as a parameter to a method but that method expects a value of the corresponding primitive type.

```
public class Example1{  
    public static void myMethod(int num)  
    {        System.out.println(num);    }  
    public static void main(String[] args)    {
```

```
Integer intObject = new Integer(100);
```

```
// passed Integer wrapper class object, would be converted to int primitive type
```

```
myMethod(intObject);    }}
```

Collections -Autounboxing

```
class Demo
{
    public static void main(String[] args)
    {
        ArrayList arrayList = new ArrayList();
        arrayList.add(100); // autoboxing int to Integer
        arrayList.add(200);
        arrayList.add(300);
        for(Integer i : arrayList) {
            System.out.println(i);
        }
        // unboxing Integer to int type
        int first = arrayList.get(0);
        System.out.println("int value "+first);
    }
}
```

Varargs

- [In a java method](#), you are not sure how many arguments your method is going to accept. To address this problem, Java 1.5 introduced varargs.
- Varargs is a short name for variable arguments.
- In Java, an argument of a method can accept arbitrary number of values.
- This argument that can accept variable number of values is called varargs.

Example

```
public int sumNumber(int ... args){  
    System.out.println("argument length: " +  
args.length);  
    int sum = 0;  
    for(int x: args){  
        sum += x;  
    }  
    return sum;  
}
```

Example

```
public static void main( String[] args ) {  
    VarargExample ex = new VarargExample();  
  
    int sum2 = ex.sumNumber(2, 4);  
    System.out.println("sum2 = " + sum2);  
  
    int sum3 = ex.sumNumber(1, 3, 5);  
    System.out.println("sum3 = " + sum3);  
  
    int sum4 = ex.sumNumber(1, 3, 5, 7);  
    System.out.println("sum4 = " + sum4);  
}
```

Enums

- An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

```
class EnumExample1{  
    //defining the enum inside the class  
    public enum Season { WINTER, SPRING, SUMMER, FALL }  
    //main method  
    public static void main(String[] args) {  
        //traversing the enum  
        for (Season s : Season.values())  
            System.out.println(s);  
    }  
}
```

Difference between Enums and Classes

- An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).
- An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

```
enum RoundingMode
{
    UP
    {
        public double round(double d)
            { return Math.ceil(d); }
    },
    DOWN
    {
        public double round(double d)
            { return Math.floor(d); }
    };
    public abstract double round(double d);
}
```