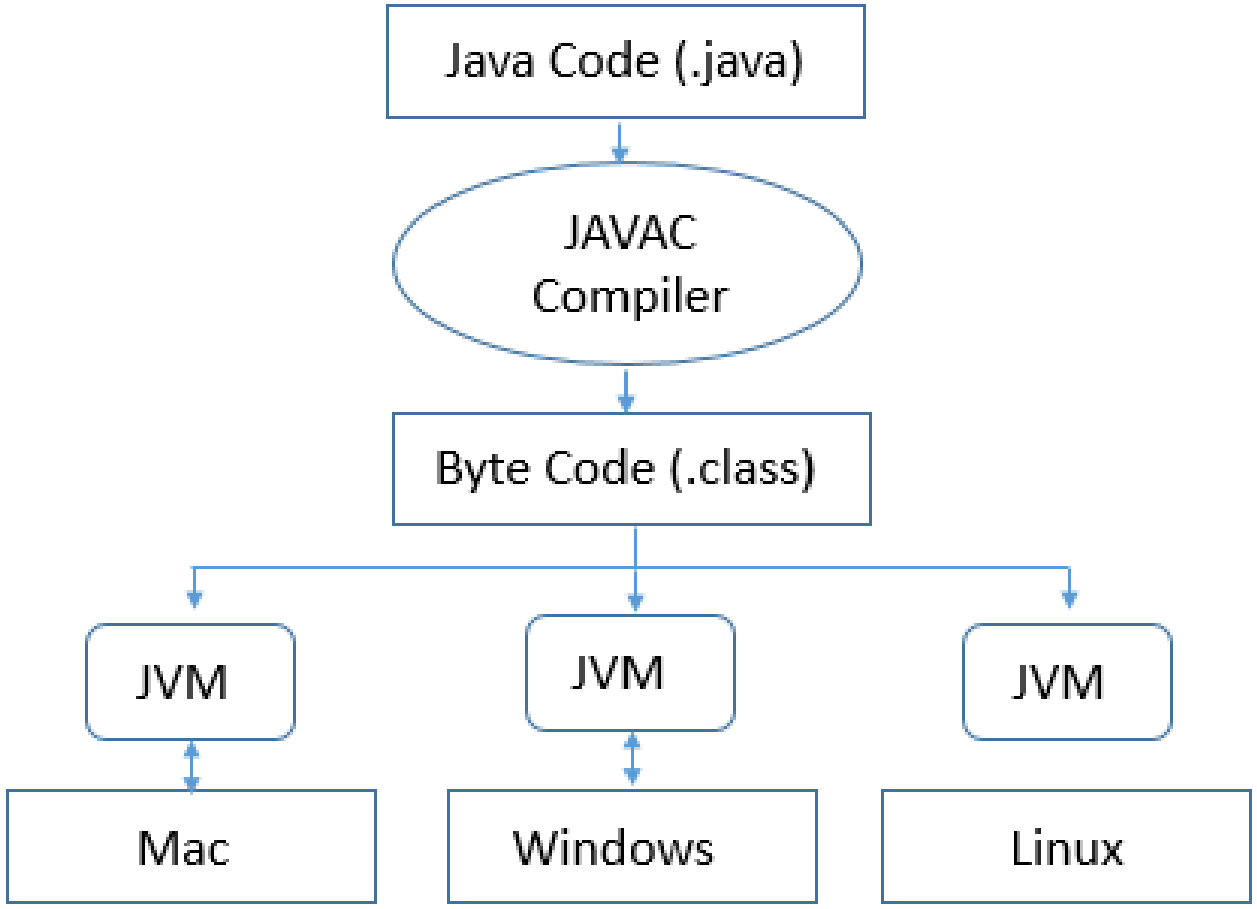
Generic Programming

Compiled by
M S Anand

Department of Computer Science

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

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, { }.

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.

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

Why Use Generics?
In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

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.

Generic Programming

Introduction

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

Introduction

Generic Types

A generic type is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

A Simple Box Class

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

Generic Programming

Introduction

A Generic Version of the Box Class
A generic class is defined with the following format:

class name<T1, T2, ..., Tn> { /* ... */ }
The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

Generic Programming

Introduction

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

Generic Programming

Introduction

Invoking and Instantiating a Generic Type

To reference the generic Box class from within your code, you must perform a generic type invocation, which replaces T with some concrete value, such as Integer:

Box<Integer> integerBox;

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a parameterized type.

To instantiate this class, use the **new** keyword, as usual, but place <Integer> between the class name and the parenthesis:

Generic Programming

Introduction

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond.

For example, you can create an instance of Box<Integer> with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Generic Programming

Introduction

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
            this.key = key;
            this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}
```

Generic Programming

Introduction

The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to autoboxing, it is valid to pass a String and an int to the class.

As mentioned in The Diamond, because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

An example:
SimpGen.java

Generic Programming

Introduction

Parameterized Types

You can also substitute a type parameter (that is, K or V) with a parameterized type (that is, List<String>). For example, using the OrderedPair<K, V> example:

```
OrderedPair<String, Box<Integer>> p = new
OrderedPair<>("primes", new Box<Integer>(...));
```

Generic Programming

Raw types

Raw Types

A raw type is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

Generic Programming

Raw types

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Generic Programming

Unchecked error messages

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

Generic Programming

Unchecked error messages

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
To completely disable unchecked warnings, use the -Xlint:-unchecked flag.
```

Generic Programming

Generic methods

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
                p1.getValue().equals(p2.getValue());
    }
}
```

Generic Programming

Generic methods

```
public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Generic Programming

Generic methods

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets

Generic Programming

Bounded type

Bounded Type Parameters

Assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to specify the type of the numbers generically, using a type parameter.

To create such a class, you might try something like this:
Stats.java

In Stats, the average( ) method attempts to obtain the double version of each number in the nums array by calling doubleValue( ). Because all numeric classes, such as Integer and Double, are subclasses of Number, and Number defines the doubleValue( ) method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create Stats objects using only numeric types. Thus, when you try to compile Stats, an error is reported that indicates that the doubleValue( ) method is unknown.

Generic Programming

Bounded type

To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to T. Furthermore, you need some way to ensure that only numeric types are actually passed.

To handle such situations, Java provides bounded types. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an extends clause when specifying the type parameter, as shown here:
<T extends superclass>

This specifies that T can only be replaced by superclass, or subclasses of superclass. Thus, superclass defines an inclusive, upper limit.

BoundsDemo.java

Generic Programming

Bounded type

Because the type T is now bounded by Number, the Java compiler knows that all objects of type T can call doubleValue( ) because it is a method declared by Number. This is, by itself, a major advantage.

However, as an added bonus, the bounding of T also prevents nonnumeric Stats objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try recompiling, you will receive compile-time errors because String is not a subclass of Number.

In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first.

Generic Programming

Bounded type

When a bound includes an interface type, only type arguments that implement that interface are legal.

When specifying a bound that has a class and an interface, or multiple interfaces, use the & operator to connect them.

For example, class Gen<T extends MyClass & MyInterface> { // ...

Here, T is bounded by a class called MyClass and an interface called MyInterface.

Thus, any type argument passed to T must be a subclass of MyClass and implement MyInterface

Generic Programming

Using Wildcard arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs.

For example, given the Stats class shown already, assume that you want to add a method called sameAvg( ) that determines if two Stats objects contain arrays that yield the same average, no matter what type of numeric data each object holds.

For example, if one object contains the double values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same.

One way to implement sameAvg( ) is to pass it a Stats argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same.

Generic Programming
Using Wildcard arguments

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
        System.out.println("Averages are the same.");
else
        System.out.println("Averages differ.");
```

At first, creating sameAvg( ) seems like an easy problem. Because Stats is generic and its average( ) method can work on any type of Stats object, it seems that creating sameAvg( ) would be straightforward.

Unfortunately, trouble starts as soon as you try to declare a parameter of type Stats. Because Stats is a parameterized type, what do you specify for Stats' type parameter when you declare a parameter of that type?

Generic Programming
Using Wildcard arguments

We might think of a solution like this, in which T is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
boolean sameAvg(Stats<T> ob) {
if(average() == ob.average())
            return true;
return false;
}
```

The trouble with this attempt is that it will work only with other Stats objects whose type is the same as the invoking object.

For example, if the invoking object is of type Stats<Integer>, then the parameter ob must also be of type Stats<Integer>. It can't be used to compare the average of an object of type Stats<Double> with the average of an object of type Stats<Short>, for example.

Therefore, this approach won't work except in a very narrow context and does not yield a general (that is, generic) solution.

Generic Programming

Using Wildcard arguments

To create a generic sameAvg( ) method, you must use another feature of Java generics: the wildcard argument.

The wildcard argument is specified by the ?, and it represents an unknown type. Using a wildcard, here is one way to write the sameAvg( ) method:

```
// Determine if two averages are the same.
// Notice the use of the wildcard.

boolean sameAvg(Stats<?> ob) {
if(average() == ob.average())
            return true;
return false;
}
```

Here, Stats<?> matches any Stats object, allowing any two Stats objects to have their averages compared.

Generic Programming
Using Wildcard arguments

The full program is WildcardDemo.java

One last point:
It is important to understand that the wildcard does not affect what type of Stats objects can be created. This is governed by the extends clause in the Stats declaration. The wildcard simply matches any valid Stats object.

Generic Programming

Bounded wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let's work through an example.

Consider the following hierarchy of classes that encapsulate coordinates:

Coordinates.java

At the top of the hierarchy is TwoD, which encapsulates a two-dimensional, XY coordinate. TwoD is inherited by ThreeD, which adds a third dimension, creating an XYZ coordinate. ThreeD is inherited by FourD, which adds a fourth dimension (time), yielding a four-dimensional coordinate

Generic Programming

Bounded wildcards

Shown next is a generic class called Coords, which stores an array of coordinates:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD>
{
        T[] coords;
        Coords(T[] o)
        {
                coords = o;
        }
}
```

Notice that Coords specifies a type parameter bounded by TwoD. This means that any array stored in a Coords object will contain objects of type TwoD or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the coords array of a Coords object. Because all types of Coords objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown in the next slide:

Generic Programming

Bounded wildcards

```
static void showXY(Coords<?> c)
{
        System.out.println("X Y Coordinates:");
        for(int i=0; i < c.coords.length; i++)
                System.out.println(c.coords[i].x + " " + c.coords[i].y);
        System.out.println();
}
```

Because Coords is a bounded generic type that specifies TwoD as an upper bound, all objects that can be used to create a Coords object will be arrays of type TwoD, or of classes derived from TwoD. Thus, showXY( ) can display the contents of any Coords object.

Generic Programming

Bounded wildcards

However, what if you want to create a method that displays the X, Y, and Z coordinates of a ThreeD or FourD object?

The trouble is that not all Coords objects will have three coordinates, because a Coords<TwoD> object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for Coords<ThreeD> and Coords<FourD> objects, while preventing that method from being used with Coords<TwoD> objects?

The answer is the bounded wildcard argument.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an extends clause in much the same way it is used to create a bounded type.

Generic Programming

Bounded wildcards

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a Coords object, if that object actually has those three coordinates.

For example, the following showXYZ( ) method shows the X, Y, and Z coordinates of the elements stored in a Coords object, if those elements are actually of type ThreeD (or are derived from ThreeD):

```
static void showXYZ(Coords<? extends ThreeD> c)
{
        System.out.println("X Y Z Coordinates:");
        for(int i=0; i < c.coords.length; i++)
                System.out.println(c.coords[i].x + " " +
                        c.coords[i].y + " " + c.coords[i].z);
        System.out.println();
}
```

Generic Programming
Bounded wildcards

Notice that an extends clause has been added to the wildcard in the declaration of parameter c. It states that the ? can match any type as long as it is ThreeD, or a class derived from ThreeD.

Thus, the extends clause establishes an upper bound that the ? can match.
Because of this bound, showXYZ( ) can be called with references to objects of type Coords<ThreeD> or Coords<FourD>, but not with a reference of type Coords<TwoD>.

Attempting to call showXZY( ) with a Coords<TwoD> reference results in a compile-time error, thus ensuring type safety.

The program: BoundedWildcard.java

Generic Programming

Bounded wildcards

Because tdlocs is a Coords(TwoD) object, it cannot be used to call showXYZ( ) or showAll( ) because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program.
You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:
<? extends superclass>
where superclass is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by superclass) is also within bounds.

You can also specify a lower bound for a wildcard by adding a super clause to a wildcard declaration. Here is its general form:
<? super subclass>
In this case, only classes that are superclasses of subclass are acceptable arguments. This is an inclusive clause.

Generic Programming

Multiple bounds

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generic Programming

Generic methods

Creating a Generic Method

As the preceding examples have shown, 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.

The following program declares a non-generic class called GenMethDemo and a static generic method within that class called isIn( ). The isIn( ) method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

Generic Programming

Generic methods …

GenMethDemo.java

Examine isIn( ) closely.

First, notice how it is declared by this line:

static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {

The type parameters are declared before the return type of the method.

Also note that T extends Comparable<T>.

Comparable is an interface declared in java.lang. A class that implements Comparable defines objects that can be ordered. Thus, requiring an upper bound of Comparable ensures that isIn( ) can be used only with objects that are capable of being compared. Comparable is generic, and its type parameter specifies the type of objects that it compares.

Next, notice that the type V is upper-bounded by T. Thus, V must either be the same as type T, or a subclass of T. This relationship enforces that isIn( ) can be called only with arguments that are compatible with each other. Also notice that isIn( ) is static, enabling it to be called independently of any object. Generic methods can be either static or non-static. There is no restriction in this regard.

Generic Programming

Generic methods …

Now, notice how isIn( ) is called within main( ) by use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the types of T and V are adjusted accordingly.

For example,
in the first call:
if(isIn(2, nums))
the type of the first argument is Integer (due to autoboxing), which causes Integer to be substituted for T. The base type of the second argument is also Integer, which makes Integer a substitute for V, too.

In the second call, String types are used, and the types of T and V are replaced by String.

Generic Programming

Generic methods …

Here is the syntax for a generic method:

<type-param-list > ret-type meth-name (param-list) { // …

In all cases, type-param-list is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic constructors

Generic Constructors

It is possible for constructors to be generic, even if their class is not.

For example, consider the following short program:
GenConsDemo.java

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.

Generic Programming

Generic interfaces

Generic interfaces are specified just like generic classes.

Here is an example. GenIFDemo.java

It creates an interface called MinMax that declares the methods min( ) and max( ), which are expected to return the minimum and maximum value of some set of objects.

A few points to note:

First, notice that MinMax is declared like this:

interface MinMax<T extends Comparable<T>> {

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is T, and its upper bound is Comparable. As explained earlier, Comparable is an interface defined by java.lang that specifies how objects are compared. Its type parameter specifies the type of the objects being compared.

Generic Programming

Generic interfaces

Next, MinMax is implemented by MyClass.
Notice the declaration of MyClass, shown here:

```
class MyClass<T extends Comparable<T>> implements
MinMax<T> {
```

Pay special attention to the way that the type parameter T is declared by MyClass and then passed to MinMax.
Because MinMax requires a type that implements Comparable, the implementing class (MyClass in this case) must specify the same bound.
Furthermore, once this bound has been established, there is no need to specify it again in the implements clause.
In fact, it would be wrong to do so.

Generic Programming

Generic Interface …

For example, this line is incorrect and won't compile:

```
// This is wrong!
class MyClass<T extends Comparable<T>>
implements MinMax<T extends Comparable<T>> {
```

Once the type parameter has been established, it is simply passed to the interface without further modification.

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface.

Generic Programming

Generic Interface …

For example, the following attempt to declare MyClass is in error:

class MyClass implements MinMax<T> { // Wrong!

Because MyClass does not declare a type parameter, there is no way to pass one to MinMax.
In this case, the identifier T is simply unknown, and the compiler reports an error.

Of course, if a class implements a specific type of generic interface, such as shown here:

class MyClass implements MinMax<Integer> { // OK
then the implementing class does not need to be generic.

Generic Programming

Generic Interface …

The generic interface offers two benefits.

First, it can be implemented for different types of data.

Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented.

In the MinMax example, only types that implement the Comparable interface can be passed to T.

Here is the generalized syntax for a generic interface:
interface interface-name<type-param-list> { // …
Here, type-param-list is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:
class class-name<type-param-list>
implements interface-name<type-arg-list> {

Generic Programming

Raw type

Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. Even today there is still pre-generics legacy code that must remain both functional and compatible with generics.

Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a raw type for the class. This raw type is compatible with legacy code, which has no knowledge of generics.

The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:

RawDemo.java

Generic Programming

Raw Type

This program contains several interesting things. First, a raw type of the generic Gen class is created by the following declaration:
Gen raw = new Gen(new Double(98.6));
Notice that no type arguments are specified. In essence, this creates a Gen object whose type T is replaced by **Object**.
A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of Gen object. The reverse is also allowed; a variable of a specific Gen type can be assigned a reference to a raw Gen object.
However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.
This lack of type safety is illustrated by the commented-out lines at the end of the program. Let's examine each case. First, consider the following situation:
// int i = (Integer) raw.getob(); // run-time error
In this statement, the value of ob inside raw is obtained, and this value is cast to Integer. The trouble is that raw contains a Double value, not an integer value. However, this cannot be detected at compile time because the type of raw is unknown. Thus, this statement fails at run time.

Generic Programming

Raw Type …

The next sequence assigns to a strOb (a reference of type Gen<String>) a reference to a raw Gen object:

```
strOb = raw; // OK, but potentially wrong
// String str = strOb.getob(); // run-time error
```

The assignment, itself, is syntactically correct, but questionable. Because strOb is of type Gen<String>, it is assumed to contain a String. However, after the assignment, the object referred to by strOb contains a Double. Thus, at run time, when an attempt is made to assign the contents of strOb to str, a run-time error results because strOb now contains a Double. Thus, the assignment of a raw reference to a generic reference bypasses the typesafety mechanism.

The following sequence inverts the preceding case:

```
raw = iOb; // OK, but potentially wrong
// d = (Double) raw.getob(); // run-time error
```

Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, raw now refers to an object that contains an Integer object, but the cast assumes that it contains a Double. This error cannot be prevented at compile time. Rather, it causes a run-time error.

Generic Programming

Raw Type …

Because of the potential for danger inherent in raw types, javac displays unchecked warnings when a raw type is used in a way that might jeopardize type safety. In the preceding program, these lines generate unchecked warnings:

Gen raw = new Gen(new Double(98.6));
strOb = raw; // OK, but potentially wrong

In the first line, it is the call to the Gen constructor without a type argument that causes the warning. In the second line, it is the assignment of a raw reference to a generic variable that generates the warning.

At first, you might think that this line should also generate an unchecked warning, but it does not:

raw = iOb; // OK, but potentially wrong

No compiler warning is issued because the assignment does not cause any further loss of type safety than had already occurred when raw was created.

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

Generic Programming

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class.

Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.

This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass

SuperClass.java

Generic Programming

Generic Class Hierarchies…

In this hierarchy, Gen2 extends the generic class Gen. Notice how Gen2 is declared by the following line:
class Gen2<T> extends Gen<T> {

The type parameter T is specified by Gen2 and is also passed to Gen in the extends clause.
This means that whatever type is passed to Gen2 will also be passed to Gen. For example, this declaration,
Gen2<Integer> num = new Gen2<Integer>(100);
passes Integer as the type parameter to Gen. Thus, the ob inside the Gen portion of Gen2 will be of type Integer.

Notice also that Gen2 does not use the type parameter T except to support the Gen superclass. Thus, even if a subclass of a generic superclass would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic superclass.

Generic Programming

Generic Class Hierarchies…

A subclass is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which Gen2 adds a type parameter of its own:

HierDemo.java

Notice the declaration of this version of Gen2, which is shown here:
class Gen2<T, V> extends Gen<T> {

Here, T is the type passed to Gen, and V is the type that is specific to Gen2. V is used to declare an object called ob2, and as a return type for the method getob2( ).

In main( ), a Gen2 object is created in which type parameter T is String, and type parameter V is Integer. The program displays the following, expected, result:

Value is: 99

Generic Programming

Generic Subclass

It is perfectly acceptable for a non-generic class to be the superclass of a generic subclass.

An example: HierDemo2.java

In the program, notice how Gen inherits NonGen in the following declaration:
class Gen<T> extends NonGen {

Because NonGen is not generic, no type argument is specified. Thus, even though Gen declares the type parameter T, it is not needed by (nor can it be used by) NonGen.

Thus, NonGen is inherited by Gen in the normal way. No special conditions apply.

Generic Programming

Run-time type comparison within a generic hierarchy

instanceof determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The **instanceof** operator can be applied to objects of generic classes.

The following class demonstrates some of the type compatibility implications of a generic hierarchy:

HierDemo3.java

Generic Programming

Run-time type comparison within a generic hierarchy

In this program, Gen2 is a subclass of Gen, which is generic on type parameter T.
In main( ), three objects are created. The first is iOb, which is an object of type Gen<Integer>.
The second is iOb2, which is an instance of Gen2<Integer>. Finally, strOb2 is an object of type Gen2<String>.

Then, the program performs these instanceof tests on the type of iOb2:

```
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
System.out.println("iOb2 is instance of Gen2");
// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
System.out.println("iOb2 is instance of Gen");
```

Generic Programming

Run-time type comparison within a generic hierarchy

As the output shows, both succeed. In the first test, iOb2 is checked against Gen2<?>. This test succeeds because it simply confirms that iOb2 is an object of some type of Gen2 object.
The use of the wildcard enables instanceof to determine if iOb2 is an object of any type of Gen2. Next, iOb2 is tested against Gen<?>, the superclass type. This is also true because iOb2 is some form of Gen, the superclass.

The next few lines in main( ) show the same sequence (and same results) for strOb2.
Next, iOb, which is an instance of Gen<Integer> (the superclass), is tested by these lines:

```
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
System.out.println("iOb is instance of Gen2");
// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
System.out.println("iOb is instance of Gen");
```

Generic Programming

Run-time type comparison within a generic hierarchy

The first if fails because iOb is not some type of Gen2 object. The second test succeeds because iOb is some type of Gen object.

Now, look closely at these commented-out lines:

```
// The following can't be compiled because
// generic type info does not exist at run time.
// if(iOb2 instanceof Gen2<Integer>)
// System.out.println("iOb2 is instance of Gen2<Integer>");
```

As the comments indicate, these lines can’t be compiled because they attempt to compare iOb2 with a specific type of Gen2, in this case, Gen2<Integer>.

Remember, there is no generic type information available at run time. Therefore, there is no way for instanceof to know if iOb2 is an instance of Gen2<Integer> or not.

Generic Programming

Casting

Casting

You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same.

For example, assuming the foregoing program, this cast is legal:

(Gen<Integer>) iOb2 // legal
because iOb2 includes an instance of Gen<Integer>.

But, this cast:
(Gen<Long>) iOb2 // illegal
is not legal because iOb2 is not an instance of Gen<Long>.

Generic Programming

Overriding methods in Generic class

A method in a generic class can be overridden just like any other method.

For example, consider this program in which the method getob( ) is overridden:

OverrideDemo.java

As the output confirms, the overridden version of getob( ) is called for objects of type Gen2, but the superclass version is called for objects of type Gen.

Generic Programming

Erasure

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of erasure.

Generic Programming

Erasure

In general, here is how erasure works.

When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is Object if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments.

The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time.

They are simply a source-code mechanism

Generic Programming

Erasure

Bridge Methods

Occasionally, the compiler will need to add a bridge method to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass.

In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass.

Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

BridgeDemo.java

Generic Programming

Erasure

In the program, the subclass Gen2 extends Gen, but does so using a String-specific version of Gen, as its declaration shows:

```
class Gen2 extends Gen<String> {
```

Furthermore, inside Gen2, getob( ) is overridden with String specified as the return type:

```
// A String-specific override of getob().
String getob()
{
        System.out.print("You called String getob(): ");
        return ob;
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of getob( ) will be Object getob() { // ...

Generic Programming

Erasure

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the String version. Thus, if you examine the class file for Gen2 by using javap, you will see the following methods:

```
class Gen2 extends Gen<java.lang.String>
{
        Gen2(java.lang.String);
        java.lang.String getob();
        java.lang.Object getob(); // bridge method
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by javap, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about this example. Notice that the only difference between the two getob( ) methods is their return type. Normally, this would cause an error, but because this does not occur in your source code, it does not cause a problem and is handled correctly by the JVM.

Note: javap disassembles a class file

Generic Programming

Ambiguity errors

The inclusion of generics gives rise to a new type of error that you must guard against: **ambiguity**.

Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on overloaded methods.
class MyGenClass<T, V>
{
        T ob1;
        V ob2;
        // ...
        // These two overloaded methods are ambiguous and will not compile.
        void set(T o) {
        ob1 = o;
        }
        void set(V o) {
        ob2 = o;
        }
}
```

Generic Programming

Ambiguity errors

Notice that MyGenClass declares two generic types: T and V.

Inside MyGenClass, an attempt is made to overload set( ) based on parameters of type T and V. This looks reasonable because T and V appear to be different types.

However, there are two ambiguity problems here.

First, as MyGenClass is written, there is no requirement that T and V actually be different types.

For example, it is perfectly correct (in principle) to construct a MyGenClass object as shown here:
MyGenClass<String, String> obj = new MyGenClass<String, String>()

In this case, both T and V will be replaced by String. This makes both versions of set( ) identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of set( ) reduces both versions to the following:
void set(Object o) { // ...
Thus, the overloading of set( ) as attempted in MyGenClass is inherently ambiguous.

Generic Programming

Ambiguity errors

Ambiguity errors can be tricky to fix.

For example, if you know that V will always be some type of Number, you might try to fix MyGenClass by rewriting its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!
```

This change causes MyGenClass to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

This works because Java can accurately determine which method to call.

Generic Programming

Ambiguity errors

However, ambiguity returns when you try this line:

MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();

In this case, since both T and V are Number, which version of set( ) is to be called? The call to set( ) is now ambiguous.

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload set( ).

Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

Generic Programming

Some generic restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays.

Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T>
{
        T ob;
        Gen()
        {
                ob = new T(); // Illegal!!!
        }
}
```

Here, it is illegal to attempt to create an instance of T. The reason should be easy to understand: the compiler does not know what type of object to create. T is simply a placeholder.

Generic Programming

Some generic restrictions

Restrictions on Static Members

No static member can use a type parameter declared by the enclosing class.

For example, both of the static members of this class are illegal:

```
class Wrong<T>
{
        // Wrong, no static variables of type T.
        static T ob;
        // Wrong, no static method can use T.
        static T getob()
        {
                return ob;
        }
}
```

Although you can't declare static members that use a type parameter declared by the enclosing class, you can declare static generic methods, which define their own type parameters (discussed earlier)

Generic Programming

Some generic restrictions

Generic Array Restrictions

There are two important generics restrictions that apply to arrays.

First, you cannot instantiate an array whose element type is a type parameter.

Second, you cannot create an array of type-specific generic references.

The following short program shows both situations:
GenArrays.java

Generic Programming

Some generic restrictions

As the program shows, it's valid to declare a reference to an array of type T, as this line does:
T vals[]; // OK

But, you cannot instantiate an array of T, as this commented-out line attempts:
// vals = new T[10]; // can't create an array of T

The reason you can't create an array of T is that there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to Gen( ) when an object is created and assign that reference to vals, as the program does in this line:
vals = nums; // OK to assign reference to existent array

This works because the array passed to Gen has a known type, which will be the same type as T at the time of object creation.

Generic Programming

Some generic restrictions

Inside main( ), notice that you can't declare an array of references to a specific generic type.

That is, this line
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
won't compile.

You can create an array of references to a generic type if you use a wildcard, however, as shown here:
Gen<?> gens[] = new Gen<?>[10]; // OK

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Programming

Some generic restrictions

Generic Exception Restriction

A generic class cannot extend Throwable.

This means that you cannot create generic exception classes

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

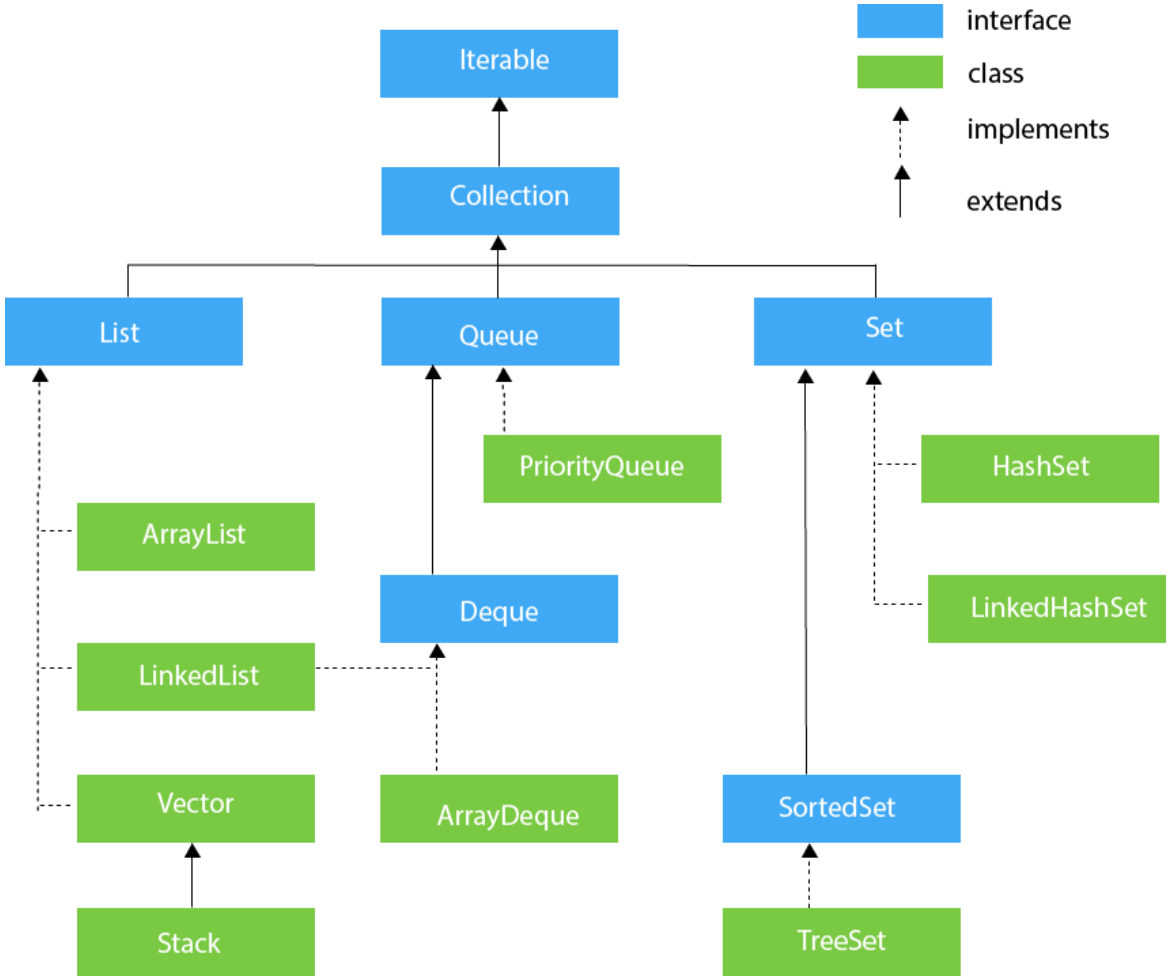
Collection framework

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

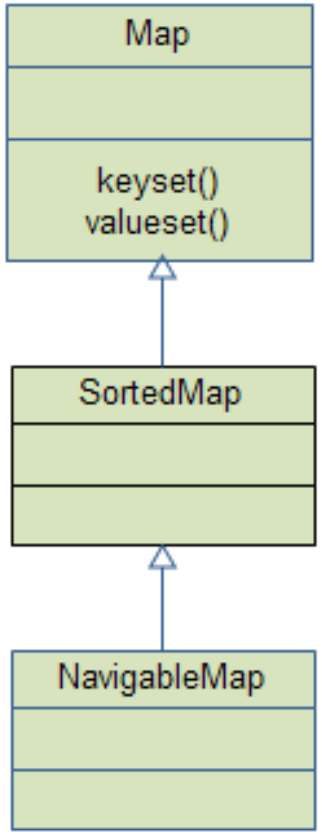
There are two "groups" of interfaces: Collections and Maps.



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.

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

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu