310-035

Sun Certified Programmer for Java 2 Platform 1.4

Non-Static Member Class in a static Contex.

```
class Cls2{
     public static class Cls2_1s{
          public class Cls2_1s_1{

          }
     }
}

class Main{
     public static void main(String[] a){
          Cls2.Cls2_1s.Cls2_1s_1 c = new Cls2.Cls2_1s.Cls2_1s_1
();
     //Cls2_1s_1 is declared in a static context so has no
enclosing instance
     }
}
```

Compilation Error:

```
an enclosing instance that contains Cls2.Cls2_1s.Cls2_1s_1 is
required
     Cls2.Cls2_1s.Cls2_1s_1 c = new Cls2.Cls2_1s.Cls2_1s_1();
```

Solution: >> provide an enclosing instance

```
class Main{
     public static void main(String[] a){
          Cls2 obj2 = new Cls2();
          Cls2.Cls2_1s obj2_1s = obj2. new Cls2_1s();

          Cls2.Cls2_1s.Cls2_1s_1 c = obj2_1s. new Cls2_1s_1();
     }
}
```

Importing Static Member Classes

File: file1.java

```
package scjp;
class Cls2{
     public static class Cls2_1s{
          public static class Cls2_1s_1s{

          }
          public interface Intf2_1s_1s{

          }
     }
}
```

File: file2.java

```
import scjp.Cls2.Cls2_1s.*
```

Private members of Member Classes Accessible in the Outer Class(using the required references)

```
class Cls2{
     public class Cls2_1s{
          private int field1;
          private int field2;
          public class Cls2_1s_1{
               private int field2;
               public void funct(){
                    System.out.println(field1);
               }
               private void funct1(){
                    System.out.println("field1:  "+field1);
                    System.out.println("field2:  "+Cls2_1s.this.
field1);
                    //Accessing Private field of the Outer Class
               }
               public class Cls2_1s_1_1{

               }
          }
          public void funct2(){
               Cls2_1s_1 obj = new Cls2_1s_1();
               obj.funct1();
               //Accessing Private field of one of the Member
Class
     //even can access private members in the Member Class of the
Member Class
          }
     }
}
```

```
obj.funct1();
```

Without obj, the compiler will not identify the funct1()

Nested Interfaces:
1. There can only be static Nested Interfaces.
Why?
Where do we use Interface?

- Interface keeps method declarations ==> For Implementation of these
- final static constants
- No instantiation is possible

All these facilities don't require binding with the other class instance. So, The nested Interfaces are always Nested.

Y Static?
Same definition for all the instatnces of a Class.

Y non-Static?
Different copy for different outer class object

OuterClass.NestedInterface ref = newOuterClass.NestedInterface(){ public void method(){/*definition*/}}

Access Modifier (specifier):
The access of a class/function can only be defined wrt its context.

An outer class is having package as its context. Thats y only public/default
An Inner Class is having outer class as its context. so, it can have all the access specifier

http://www.phptr.com/articles/article.asp?p=31110&seqNum=3&rl=1

acess$000(...) static fuctions of this patterns are added to the outerclass to provide access of it's private datamembers to the innerclass.

1. static member classes and interfaces
   [inner classer]
   a. on-static member classes
   b. local classes
   c. anonymous classes
2. An instance of an inner class can access the members of its enclosing object by their simple name.
3. A static member class or interface is defined as a static member in a class or an interface.
   Such a nested class can be instantiated like any ordinary top-level class, using its full name.
   No enclosing object is required to instantiate a static member class.
4. there are no non-static member, local, or anonymous interfaces. Interfaces are always defined either at the top level or as static members.
5. Non-static member classes are defined as instance members of other classes, just like fields and instance methods are defined in a class.
   An object of a non-static member class always has an enclosing object associated with it.
6. Local classes can be defined in the context of a block as in a method body or a local block, just as local variables can be defined in a method body or a local block.
7. Anonymous classes can be defined as expressions and instantiated on the fly.
8. An instance of a local (or an anonymous) class has an enclosing instance associated with it, if the local (or anonymous) class is declared in a non-static context.
9. A nested class or interface cannot have the same name as any of its enclosing classes or interfaces.
10. Table

Entity	Declaration Context	Accessibility Modifiers	Enclosing Instance	Direct Access to Encl Context	Declarations in Entity Body
Top-level Class (or Interface)	Package	Public or default	No	N/A	All that are valid in a class (or interface) body
Static Member Class (or Interface)	As static member of enclosing class or interface	All	No	Static members in enclosing context	All that are valid in a class (or interface) body
Non-static Member Class	As non-static member of enclosing class or interface	All	Yes	All members in enclosing context	Only non-static declarations + final static fields
Local Class	In block with non-static context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields

Local Class	In block with static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields
Anonymous Class	As expression in non-static context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields
Anonymous Class	As expression in static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields

11. Nested interfaces are implicitly static, the keyword static can, therefore, be omitted.

12. Since static member classes and interfaces are members of an enclosing class or interface, they can have any member accessibility.

13. Static member classes and interfaces can only be nested within other static member or top-level classes and interfaces.

14. A static member class can be instantiated without any reference to any object of the enclosing context, as is the case for instantiating top-level classes.

```
TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1 objRef1 =
        new TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1();
```

15. It will result in the generation of the following class files, where each file corresponds to a class or interface definition:

```
TopLevelClass$StaticMemberClass_1$StaticMemberClass_1_1.class
TopLevelClass$StaticMemberClass_1$StaticMemberInterface_1_1.class
TopLevelClass$StaticMemberClass_1.class
TopLevelClass$StaticMemberInterface_1.class
TopLevelClass.class
AnotherTopLevelClass.class
```

16. If main() is in StaticMemberClass_1_1 class then the program will be executed as java TopLevelClass$StaticMemberClass_1$StaticMemberClass_1_1

17. Importing Member Classes:
import express.TopLevelClass.*;
now inner classes can be used without qualifying the TopLevelClass

18. Accessing Members of the other member classes:
The private fields and methods of the member classes are also available to other member classes.

19. Accessing members of member classes:
The private fields and methods of the member classes are available to the enclosing class.

20. Accessing members in enclosing context: Static Member Classes
any code in a static member class can only directly access static members in its enclosing context, not instance members in its enclosing context.
Static method does not have a this reference and can only directly access other members that are declared static within the same class.
a static member class can define both static and instance members, like any other top-level class. However, its code can only directly access static members in its enclosing context.

21. Accessing members in enclosing context: Non-Static Member Classes

Code in a non-static member class can directly refer to any member (including nested) of any enclosing class or interface, including private members. No explicit reference is required.
public NonStaticMemberClass() { memberClassField = enclosingClassField; } // (9)

22. Accessing members in enclosing context: Non-Static Member Classes
public NonStaticMemberClass() { memberClassField = this.enclosingClassField; } // (9)
NOTE: this.memberClassField can't be used. NOT
Xpublic NonStaticMemberClass() { this.memberClassField = enclosingClassField; } // (9)
as the current object (indicated by this) of NonStaticMember Class has no field headlines. Its field of the enclosed object of class ToplevelClass

23. <enclosing class name>.this.enclosingClassField
evaluates to a reference that denotes the enclosing object (of class <enclosing class name>) for the current object of a non-static member class.
public NonStaticMemberClass() { this.memberClassField = EnclosingCls.this. enclosingClassField; }

24. The non-static member class does not provide any services, only instances of the class do. This implies that a non-static member class cannot have static members. However, final static variables are allowed, as these are constants.

25. Instantiating Non-static Member Classes: An object of a non-static member class can only exist with an object of its enclosing class. This means that an object of a non-static member class must be created in the context of an object of the enclosing class.
<enclosing object reference>.new <non-static member class constructor call>

26.
```
class ToplevelClass {                                   // (1)
   public NonStaticMemberClass makeInstance() {         // (3)
                  // creates an instance of the NonStaticMemberClass using the new operator
      return new NonStaticMemberClass();                // (4)
                  // creates an instance of a non-static member class in the context of the
       // instance of the enclosing class on which the makeInstance() method is invoked
      //this.new NonStaticMemberClass();
   }
```

27. ToplevelClass.NonStaticMemberClass innerRef3 =
topRef.new NonStaticMemberClass(); // (17) special form of the new operator

28. Accessing Hidden Members from the enclosing context: static context
TLClass.StaticNC.field1

29. Accessing Hidden Members from the enclosing context: non-static context
<enclosing class name>.this.fieldname;

30. Non-Static inner class is instantiated with an object of outerClass.
   a. objOuter.new InnerClass()
   b. objOuter.super() call in the constructor of the extending class (class that's extending some inner-class)

31. Extending InnerClass: mandatory to set up the proper relationships between the objects involved.
Mandatory to specify the outerClass object.

32. An instance of Outer class is explicitly passed as argument in the constructor call to InnerClass eXtending Class. The constructor of this class has a special super()

call in its body: outerRef.super();
This call ensures that the constructor of the superclass InnerA has an outer object to bind to.

a. Using the standard super() call in the subclass constructor is not adequate, because it does not provide an outer instance for the superclass constructor to bind to.
b. The non-default constructor and the outerRef.super() expression are mandatory to set up the proper relationships between the objects involved.
c.
```
class OuterA {                          // (1)
   class InnerA { }                     // (2)
}
class SomeUnrelatedClass extends OuterA.InnerA { // (3) Extends NSMC at (2)
   // (4) Mandatory non-default constructor .. MUST
   SomeUnrelatedClass(OuterA outerRef) {
      outerRef.super();                 // (5) Explicit super() call
   }
}
```
d. An effort of extending OuterClass.InnerClass without making such arrangements, results in compilation error: no enclosing instance of type OuterClass is in scope

33. The outer object problem mentioned above does not arise if the subclass that extends an inner class is also declared within an outer class that extends the outer class of the superclass.
```
class OuterB extends OuterA {              // (6) Extends class at (1)
   class InnerB extends OuterA.InnerA { }  // (7) Extends NSMC at (2)
}
```
An object of class OuterB can act as an outer object for an instance of class InnerA. The object creation expression:
new OuterB().new InnerB();

34. A local class is an inner class that is defined in a block. This block could be a method body, a constructor, a local block, a static initializer, or an instance initializer.

35. An object of a local class, which is declared in a non-static block, has an object of the enclosing class associated with it. This gives such a non-static local class much of the same capability as a non-static member class.

36. If the block containing a local class declaration is defined in a static context (i.e., a static method or a static initializer), then the local class is implicitly static in the sense that its instantiation does not require any outer object. This aspect of local classes is just like a static member classes.

37. a local class cannot be specified with the keyword static.

38. Local classes cannot have static members, as they cannot provide class-specific services. However, final static fields are allowed, as these are constants.

39. Local classes cannot have any accessibility modifier. The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations.

40. Local Class: accessing Declarations in Enclosing Block
ONLY final: final local variables, final method parameters, and final catch-block parameters

41. There is no way for the local class to refer to hidden declarations of the enclosing-block by the ones in the local class.

42. Local Class: Accessing Members in the Enclosing Class
Same as the member class

43. Clients outside the scope of a local class cannot instantiate the class directly. A local class can be instantiated in the block in which it is defined. Like a local variable, a local class must be declared before being used in the block.

44. A method can return instances of any local class it declares. The local class type must then be assignable to the return type of the method. The return type cannot be the same as the local class type, since this type is not accessible outside of the method. A supertype of the local class must be specified as the return type. This also means that, in order for the objects of the local class to be useful outside the method, a local class should implement an interface or override the behavior of its supertypes.

45. Anonymous classes are implicitly final.

46. Can have no constructors. Constructor of the super class is called.

47. **Access rules for local classes also apply to anonymous classes.**

48. Syntax when extending class

New <superclass name> **(** <optional argument list>) {<member declarations> } ;

Example

```
Public Shape createShape() {                    // (4) Non-static Method
        return new Shape(){                     // (5) Extends superclass at (2)
                Public void draw() { System.out.println("Drawing a new Shape."); }
        };<<<be careful about semi-colon ;body should end with ;
}
```

49. >>>although argument list is optional>>> () cannot be omitted

50. **The superclass must provide a constructor corresponding to the arguments passed.**

51. An instance initializer can be used to achieve the same effect as a constructor

52. **Syntax when Implementing an Interface**

New <interface name>() { <member declarations> }

class TubeLight extends Light { ç Header of the class ç extends clause

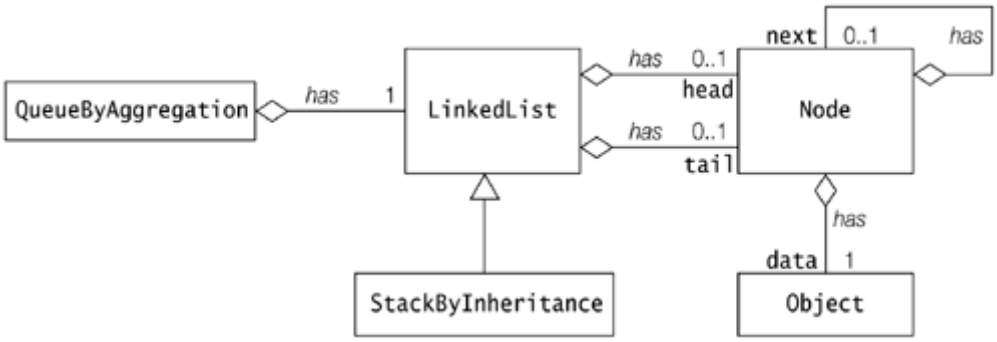
If no extends clause is specified in the header of a class declaration, then the class implicitly inherits from the java.lang.Object class. Thats the top-most class in an inheritance hierarchy chain [super1 ← super2 ← super3 ← sub]

Private members of the superclass are not inherited by the subclass and can only be indirectly accessed. but exists in the subclass object and is indirectly accessible.
the subclass can use other inherited members as if they were declared in its own class body. (by simple name or `this`)
Members that have package accessibility in the superclass are also not inherited by subclasses in other packages. These members are only accessible by their simple names in subclasses within the same package as the superclass.

Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

Inheritance defines "is-a" Relationship between subClass and superClass. This means that an object of a subclass can be used wherever an object of the superclass can be used. (As SuperClass is a SubClass) TubeLight is a Light.

Aggregation defines "has-a" Relationship between a class and its constituents. An instance of class Light has the following parts: a field to store its wattage (watts), a field to store whether it is on or off (indicator), etc. In Java, a composite object cannot contain other objects. It can only have references to its constituent objects. This relationship defines an aggregation hierarchy. Constituent objects can be shared between objects and their lifetimes can be independent of the lifetime of the composite object.



Casting:
Reference of one type is Casted into another type
Upcasting
Subclass reference is assigned to superClass reference. As an object of subclass can be used wherever an objet of super can be used.
Object objRef = stringRef;
Both references denote the same string object.
Its not possible to invoke methods exclusive to the subclass, String via the superclass reference objRef:
`objRef.length()`would be flagged as a compile-time error.
as the compiler only knows the class of the reference. It does not know what object the reference objRef is denoting.As the declaration of the Object class does not have a method length(), invocation of length() using objRef with flag a compile-time error.

Downcasting
Superclass reference is assigned to a subclass type.

by assigning references down the inheritance hierarchy, which requires explicit casting.

Compile Time Binding

1. All types of Fields
2. Private Method Calls ç Implicitly final
3. Final Method Calls
4. Static Members

are bound at the compile time. ç their output is defined by the type of reference used to access them, i.e., defined by the reference.

Polymorphism: Dynamic Method Binding

The method invoked is dependent on the type of the actual object denoted by the reference at runtime.
The actual method is determined by dynamic method lookup.
The ability of a superclass reference to denote objects of its own class and its subclasses at runtime is called polymorphism.

```
objRef = strRef;
objRef.equals("String");
```

the reference objRef refer to an object of the String class, resulting in the equals() method from the String class being executed, and not the one in the Object class.
When an instance method is invoked on an object using a reference, it is the class of the current object denoted by the reference, not the type of the reference, that determines which method implementation will be executed. << regardless of the reference type.

Casting Test @ Compile Time:

The compiler verifies that an inheritance relationship exists between the source reference type and the reference type specified in the cast.

Casting Test @ Runtime:

Whether the reference can hold the object???
However, the cast can be invalid at runtime.

1. If, at runtime, the object is cast to some unrelated class.
2. If, at runtime, object is cast to a subclass.
   subclasses can provide additional facilities, those aren't supported by object. This will result in in-consistency

a `ClassCastException` would be thrown at runtime.
The instanceof operator can be used to determine the runtime type of an object before any cast is applied.
(Rules for Instanceof are the same as for casting - if no inheritance relationship exists the it will result in compile time error)

Instance Member Overridding

A subclass may override non-static methods defined in the superclass that would otherwise be inherited. When the method is invoked on an object of the subclass, it is the new method implementation in the subclass that is executed.
The overridden method in the superclass is not inherited by the subclass, and the new method in the subclass must uphold the following rules of method overriding:

1. The new method definition must have the same method signature (i.e., method name and parameters) and the same return type.
   The parameters in the overriding method can be made final in the subclass. A method's signature does not encompass the final modifier of parameters, only their types and order.
2. The new method definition cannot narrow the accessibility of the method, but it can

widen it.

3. The new method definition can only specify all or none, or a subset of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the superclass.

The sub-class overriding method doesn't throw any exception, what if, Super-Class one can have an ability to throw an exception.

Interfaces:

These requirements also apply to interfaces, where a subinterface can override method prototypes from its superinterfaces

```
class Light {
    protected String billType = "Small bill";     // (1)

    protected double getBill(int noOfHours)
              throws Exception {      // (2)
        if (noOfHours < 0)
            throw new NegativeHoursException();
        double smallAmount = 10.0,
               smallBill = smallAmount * noOfHours;
        System.out.println(billType + ": " + smallBill);
        return smallBill;
    }
    public static void printBillType() {          // (3)
        System.out.println("Small bill");
    }
    public double getBillX() throws IOException{   // Throws an
exception
        System.out.println("No billX");
        return 0.0;
    }
}

class TubeLight extends Light {
    public static String billType = "Large bill"; // (4) Hiding
static field.

    public double getBill(final int noOfHours)
           throws IOException {     // (5) Overriding instance
method. (SubClass of Exception
        if (noOfHours == 0)
            throw new IOException();
        double largeAmount = 100.0,
               largeBill = largeAmount * noOfHours;
        System.out.println(billType + ": " + largeBill);
        return largeBill;
    }

    public static void printBillType() {          // (6) Hiding
static method.
        System.out.println(billType);
    }

    public double getBill() {                     // (7)
Overloading method.
```

```
        System.out.println("No bill");
        return 0.0;
    }
     public double getBillX() {                  //NO throws
clause
                                              //Overriding the one
from super class
        System.out.println("No billX......");
        return 0.0;
    }
}

          :
          :
        TubeLight tubeLight = new TubeLight();    // (9)
        Light     light1    = tubeLight;          // (10)
Aliases.
        Light     light2    = new Light();        // (11)

        System.out.println("Invoke overridden instance method:");
        tubeLight.getBill(5);                     // (12) Invokes
method of Tubelight
        light1.getBill(5);                        // (13) Invokes
method of Tubelight
        light2.getBill(5);                        // (14) Invokes
method of Light

       System.out.println("Access hidden field:");
        System.out.println(tubeLight.billType);   // (15)
Accesses field of Tubelight  (ref)
        System.out.println(light1.billType);      // (16)
Accesses field of Light (ref)
        System.out.println(light2.billType);      // (17)
Accesses field of Light (ref)

        System.out.println("Invoke hidden static method:");
        tubeLight.printBillType();                // (18) Invokes
method of Tubelight
        light1.printBillType();                   // (19) Invokes
method of Light
        light2.printBillType();                   // (20) Invokes
method of Light

        System.out.println("Invoke overloaded method:");
        tubeLight.getBill();                      // (21) Invokes
method at (7)
```

A subclass must use the keyword super in order to invoke an overridden method in the superclass.

A final method cannot be overridden because the modifier final prevents method overriding. An attempt to override a final method will result in a compile-time error.
A final field in the Super class can be hidden in the sub-class.

Private methods are not inherited. The method is not accessible outside the class in which it is defined; therefore, a subclass cannot override it. A subclass can give its own definition of such a method, which may have the same signature as the method in its superclass.

Static(super)~Non-static(sub)/Non-static~Static Method overriding: NO<< Compilation error
An instance method in a sublass cannot override a static method in the superclass.
A static method in a sublass cannot override an instance method in the superclass.
The compiler will flag this as an error.
A static method is class-specific and not part of any object, while overriding methods are invoked on behalf of objects of the subclass.

Method Hiding:
Static(super)~Static(sub) Method Hiding:
A static method in a subclass can hide a static method in the superclass if the exact requirements for overriding instance methods are fulfilled. A hidden superclass static method is not inherited. The compiler will flag an error if the signatures are the same but the other requirements regarding return type, throws clause, and accessibility are not met.
If the signatures are different, the method name is overloaded, not hidden.

Field Hiding:
The subclass can define fields with the same name as in the superclass. ⇒ the fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass.
Code in the subclass can use the keyword super to access such members, including hidden fields.
A client can use a reference of the superclass to access members that are hidden in the subclass.

Static field can be hidden by non-static field. NO Restrictions like the static Methods <<
As fields have compile time binding >> the reference determine which on is to be called << no concept of polymorphism

A hidden static method can be invoked by using the superclass name in the subclass declaration. The keyword super can be used in non-static code in the subclass declaration to invoke hidden static methods

Overloading:
when the method names are the same, but different signature.
to overload methods the parameters must differ in type, order, or number. As the return type is not a part of the signature, having different return types is not enough to overload methods.
A method can be overloaded in the class it is defined in or in a subclass of its class.
Invoking an overridden method in the superclass from a subclass requires special syntax (e.g., the keyword super). This is not necessary for invoking an overloaded method in the superclass from a subclass. If the right kinds of arguments are passed in the method call occurring in the subclass, the overloaded method in the superclass will be invoked.

Method Overloading Resolution : how parameter resolution is done to choose the right implementation for an overloaded method?
The most specific method is chosen.

```
class Light { /* ... */ }
class TubeLight extends Light { /* ... */ }

public class OverloadResolution {
   boolean testIfOn(Light aLight)        { return true; }    // (1)
   boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)
   public static void main(String[] args) {

      TubeLight tubeLight = new TubeLight();
      Light     light     = new Light();

      OverloadResolution client = new OverloadResolution();
      System.out.println(client.testIfOn(tubeLight));// (3) ==> method at (2)
      System.out.println(client.testIfOn(light));    // (4) ==> method at (1)
```

satisfies the parameter lists in both the implementations given at (1) and (2), as the reference tubeLight, which denotes an object of class TubeLight, can also be assigned to a reference of its superclass Light. The most specific method, (2), is chosen.

Object Reference super
The super keyword is used to access members of the super class (a level up in its inheritance hierarchy). The super can only be used in non-static code, but only in a subclass, to access fields and invoke methods from the superclass.
Method invocations with super, the method from the superclass is simply invoked regardless of whether the current class overrides the method.
Unlike the this keyword, the super keyword cannot be used as an ordinary reference. For example, it cannot be assigned to other references or cast to other reference types.
super.super.method() isn't allowed.

```
class Light {

    protected    String billType  = "Small bill";       // (1)

    protected double getBill(   int noOfHours){      // (2)
        /* ... */     }

    public static void printBillType() {             // (3)
        /* ... */     }
```

```
    public void banner() {                              // (4)
        /* ... */     }
}

class TubeLight extends Light {

    public static String billType = "Large bill";    // (5)
Hiding static field.

    public double getBill(final int noOfHours){      // (6)
Overriding instance method.
        /* ... */     }

    public static void printBillType() {             // (7)
Hiding static method.
        /* ... */     }

    public double getBill(     ) {                   // (8)
Overloading method.
        /* ... */     }

    // Inherits banner()<< can be called using this or just
name... 'n in sub-class using super.
}

class NeonLight extends TubeLight {
    // ...
    public void demonstrate(){                     // (9)

        super.banner();                              // (10)
Invokes method at (4)
        super.getBill();                             // (11)
Invokes method at (8)
        super.getBill(20);                           // (12)
Invokes method at (6)
        ((Light) this).getBill(20);                  // (13)
Invokes method at (6)
        System.out.println(super.billType);          // (14)
Accesses field at (5)
        System.out.println(((Light) this).billType); // (15)
Accesses field at (1)
        super.printBillType();                       // (16)
Invokes method at (7)
        ((Light) this).printBillType();              // (17)
Invokes method at (3)
          //Static Method
    }
}
```

Wrong Use:
super.super.doIt();
this.super.doIt();
super and this aren't part of the class, they are not inherited by a subclass.
super isn't a part of the class, so can't be accessed using this keyworld.
super and this are keywords, aren't part of the class.

object.super.msg.text
.super() << to bind an Inner Class with the other class object <<Special pattern
Compilation error

Multi-level access??? .....someField
Allowed:
object.msg.text
this.object.msg.text
super.object.msg.text

Chaining Constructors Using `this()` and `super()`
overloaded constructors
Constructors cannot be inherited or overridden. They can be overloaded.
`this()` **Constructor Call**
the this reference is used to access the fields shadowed by the parameters.
this() is used to implement local chaining of constructors in the class when an instance of the class is created. The this() call invokes the constructor with the corresponding parameter list.
any this() call must occur as the first statement in a constructor. This restriction is due to Java's handling of constructor invocation in the superclass when an object of the subclass is created.

```
class Light {
    // Fields
     /*...*/
    // Constructors
    Light() {                               // (1) Explicit
default constructor
        this(0, false);
        System.out.println("Returning from default constructor
no. 1.");
    }
    Light(int watt, boolean ind) {          // (2) Non-default
        this(watt, ind, "X");
        System.out.println("Returning from non-default
constructor no. 2.");
    }
    Light(int noOfWatts, boolean indicator, String location)
{   // (3) Non-default
        System.out.println("Returning from non-default
constructor no. 3.");
    }
}
```

`super()` **Constructor Call** >> Vertical Chaining >> chaining of subclass constructors to superclass constructors.
used in a subclass constructor to invoke a constructor in the immediate superclass (based on the signature of the call). This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created.
[super keyword can be used in a subclass constructor to access inherited/hidden members via its superclass.]

The super() construct has the same restrictions as the this() construct:
if used, the super() call must occur as the first statement in a constructor, and
it can only be used in a constructor declaration. This implies that this() and super() calls cannot both occur in the same constructor.

If a constructor has neither a this() nor a super() call as its first statement, then a super() call to the default constructor in the superclass is inserted implicitly.

This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way to the top of the inheritance hierarchy, which is always the Object class.
The body of the constructors is executed in the reverse order to the call order, as super()

can only occur as the first statement in a constructor. This ensures that the constructor from the Object class is completed first, followed by the constructors in the other classes down to the class being instantiated in the inheritance hierarchy.

If a class only defines non-default constructors, then its subclasses cannot rely on the implicit super() call being inserted. This will be flagged as a compile-time error. The subclasses must then explicitly call a superclass constructor, using the super() construct with the right arguments.

```
class SuperClass{                   //NO Default Constructor
     SuperClass(int i, int j){/*...*/}
}

class SubClass extends SuperClass {
    // Field

    Subclass() {          // (1)
        super(10, 2);     // (2) Cannot be commented out. As nO
default one in  MUST
superClass
        /*...*/
    }
    // ...
}
```

Interface:

```
<accessibility modifier> interface <interface name>
                                <extends interface clause> //
Interface header

{ // Interface body
     <constant declarations>
     <method prototype declarations>
     <nested class declarations>
     <nested interface declarations>
}
```

The methods in an interface are all implicitly `abstract` and `public`.
`abstract` by default. No need to explicitly declare it Abstract. This means that it cannot be instantiated, but classes can implement it by providing implementations for its method prototypes.
`abstract interface{ /*...*/ }` << NO cOmpilation error
interface members implicitly have `public` accessibility. Since, interface defines contracts.
interface methods cannot be declared `static`, because they comprise the contract fulfilled by the objects of the class implementing the interface.
Interfaces with empty bodies are used as **markers** to tag classes as having a certain property or behavior. Such interfaces are also called **ability interfaces.**

Implementing Interfaces:
The criteria for overriding methods also apply when implementing interface methods.
The interface methods must all have public accessibility when implemented in the class (or its subclasses). A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's throws clause, as attempting to do so would amount to altering the interface's contract, which is illegal.
Interface methods are always implemented as instance methods.

A class can choose to implement only some of the methods of its interfaces, . The class must then be declared as `abstract`.

regardless of how many interfaces a class implements directly or indirectly, it only provides a single implementation of a member that might have multiple declarations in the interfaces.

Extending Interfaces
An interface can extend other interfaces, using the `extends` clause.
Unlike extending classes, an interface can extend several interfaces.
`interface Int2 extends Int1, Int{ /*...*/}`

A subinterface inherits all methods from its superinterfaces, as their method declarations are all implicitly public. A subinterface can override method prototype declarations from its superinterfaces. Overridden methods are not inherited. Method prototype declarations can also be overloaded, analogous to method overloading in classes.

Although interfaces cannot be instantiated, references of an interface type can be declared. References to objects of a class can be assigned to references of the class' supertypes.

Constants in Interfaces

An interface can also define named constants. Such constants are defined by field declarations and are considered to be public, static and final. These modifiers can be omitted from the declaration.
Such a constant MUST be initialized with an initializer expression.

An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly without using the fully qualified name. Such a client inherits the interface constants.

Extending an interface that has constants is analogous to extending a class having static variables.
these constants can be hidden by the subinterfaces.
In the case of multiple inheritance of interface constants, any name conflicts can be resolved using fully qualified names for the constants involved.

```
interface Int{ int A = 1;}
abstract interface Int1{int A = 2;}
abstract interface Int2 extends Int1, Int{int B=1;}
class Cls2 extends Cls implements Int2{
     {    //System.out.println(A);
   //reference to A is ambiguous, both variable A in Int1 and
variable A in Int match
          System.out.println(Int.A);//fully qualified names for
the constants
     }
}
--------------
interface MyConstants {
    int r = 42;
    int s = 69;
    // INSERT CODE HERE
}
```

ü `final double circumference = 2*Math.PI*r;`
û `int total = total + r + s;` `//illegal forward reference to its own named constant`
ü `int AREA = r*s;`
û `public static ___ MAIN = 15;` `//field type is missing`
û ~~`protected`~~ `int CODE = 31337;`

Class, interface, and array types >> (reference types)

1. All reference types are subtypes of Object type.
2. All arrays of reference types are also subtypes of the array type Object[], but arrays of primitive data types are not. Note that the array type Object[] is also a subtype of Object type.
3. If a reference type is a subtype of another reference type, then the corresponding array types also have an analogous subtype-supertype relationship. SuperClass <- SubType => SuperClass[] <- SubType[]
4. There is no subtype-supertype relationship between a type and its corresponding array type. SuperClass <-x-> SuperClass[]

We can create an array of an interface type, but we cannot instantiate an interface (as is the case with abstract classes).
Sup[] iSupArray = new Sup[5]; // Array of Interface
//The array creation expression creates an array whose element type is ISup.
An array reference exhibits polymorphic behavior like any other reference.

SuperCls[] SuperClsArray = new SubCls[5];
a supertype reference (SuperCls[]) can denote objects of its subtype (SubCls[])

SuperClsArray[0] = new SubCls(); // SuperCls is a supertype of SubCls

-->
SuperClsArray[0] = new SuperCls(); // ArrayStoreException **ç** SuperClsArray[0] is a reference of type SubCls
Compile Time - no problems, As compiler cannot deduce that the array variable SuperClsArray will actually denote an SubCls[] object at runtime. **ç**

ArrayStoreException to be thrown at runtime, as a SubCls REFERENCES cannot possibly contain objects of type SuperCls.

Ref 1 lvl	**ü**	Ref 2 lvl	**ü**	**û**
SuperCls[]	SubCls[] NO PROBLEM	subCls[0]	new SubCls()	~~new SuperCls()~~
		subCls[1]		Compile time NO PROBLEM Run-time ArrayStoreExcep SuperClass Obj can't be assigned to subClass ref.
		subCls[2]		
		subCls[3]		
		subCls[4]		
		subCls[5]		

Assigning, Passing, and Casting Reference Values:

For values of the primitive data types and reference types, conversions can occur during
- assignment
- parameter passing
- explicit casting

The rule of thumb for the primitive data types is that widening conversions are permitted, but narrowing conversions require an explicit cast.
The rule of thumb for reference values is that conversions up the type hierarchy are permitted (upcasting), but conversions down the hierarchy require explicit casting (downcasting).
subtype to its supertypes are allowed, other conversions require an explicit cast or are illegal.

The rules for reference value assignment:

```
SourceType srcRef;
// srcRef is appropriately initialized.
DestinationType destRef = srcRef;
```

If an assignment is legal, then the reference value of srcRef is said to be assignable (or assignment compatible) to the reference of DestinationType.
The rules for assignment are enforced at compile time, guaranteeing that no type conversion error will occur during assignment at runtime. Such conversions are type safe. The reason the rules can be enforced at compile time is that they concern the type of the reference (which is always known at compile time) rather than the actual type of the object being referenced (which is known at runtime).
The rules are -

1. If SourceType is a class type, then the reference value in srcRef may be assigned to the destRef reference, provided DestinationType is one of the following:
   - DestinationType is a superclass of the subclass SourceType.
   - DestinationType is an interface type that is implemented by the class SourceType.

     objRef = subClassRef; // (1) Always possible
     superClassRef = subClassRef; // (2) Subclass to superclass assignment
     iSuperTypeRef = superClassRef; // (3) SuperClass implements IntefacesSuperType
     iSubTypeRef = subClassRef; // (4) SubClass implements InterfaceSubType
2. If SourceType is an interface type, then the reference value in srcRef may be assigned to the destRef reference, provided DestinationType is one of the following:
   - DestinationType is Object.
   - DestinationType is a superinterface of subinterface SourceType.

     objRef = iSubTypeRef; // (5) Always possible
     iSuperTypeRef = iSubTypeRef; // (6) Subinterface to superinterface assignment
3. If SourceType is an array type, then the reference value in srcRef may be assigned to the destRef reference, provided DestinationType is one of the following:
   - DestinationType is Object.
   - DestinationType is an array type, where the element type of SourceType is assignable to the element type of DestinationType.

     objRef = objArray; // (7) Always possible Object[] to Object

```
objRef      = superClassArray;          // (8) Always possible       anyClass[] to
Object
objArray            = superClassArray;          // (9) Always possible
anyClass[] to Object[]
objArray            = iSuperTypeArray; // (10) Always possible      anyInterface[]
to Object[]
objRef      = intArray;                 // (11) Always possible      Object to int[]
// objArray         = intArray;                 // (12) Compile-time error
superClassArray = subClassArray;    // (13) Subclass array to superclass array
    subClass[] to superClass[]
iSubTypeArray    = subClassArray;    // (14) SubClass implements
InterfaceSubType        interface[] to implClass[]
```

Parameter Passing Conversions
The rules for reference value assignment conversion also apply for parameter passing conversions.

Reference Casting and instanceof Operator
The expression to cast a <reference> of <source type> to <destination type> has the following syntax:
(<destination type>) <reference>

checks that the reference value of the object denoted by the <reference> is assignable to a reference of the <destination type>,--> <source type> is compatible to the <destination type>. If this is not the case, a ClassCastException is thrown.

The null reference value can be cast to any reference type.

The binary instanceof operator has the following syntax (note that the keyword is composed of only lowercase letters):
<reference> instanceof <destination type>

The instanceof operator returns true if the left-hand operand (<reference>) can be cast to the right-hand operand (<destination type>).
Always returns false if the left-hand operand is null.
(null instanceof Object)   // Always false.
If the instanceof operator returns true, then the corresponding cast expression will always be valid.

Both the cast and the instanceof operators require a compile-time check and a runtime check:
The compile-time check determines
whether <source type> and <destination type> are in the type hierarchy.
[<source type> and <destination type> reference type can denote objects of a reference type that is a common subtype of both <source type> and <destination type> in the type hierarchy.]
If this is not the case, then obviously there is no relationship between the types, and neither the cast nor the instanceof operator application would be valid.
The runtime check -
it is the type of the actual object denoted by the <reference> that determines the outcome of the operation.
<source Object type> assignable to <destination type>
<source Object type> is subType of <destination type> in the type hierarchy .

if Not Assignable -
instanceof -> false
casting -> ClassCastException

Converting References of Class and Interface Types
References of an interface type can denote objects of classes that implement this interface. ç upcasting.
converting a reference value of interface type to the type of the class implementing the interface, ç downcasting. requires explicit casting.

```
InterfaceSuperType iSuperTypeRef = new SuperClass(5);
// Upcasting [SuperClass implements InterfaceSuperType] <-
SubClass
SuperClass superClassRef  = (SuperClass) iSuperTypeRef;       //
Downcasting
```

Using the reference iSuperTypeRef of interface type InterfaceSuperType, methods of the InterfaceSuperType interface can be invoked on objects of the SuperClass class that implements this interface. However, the additional members of the SuperClass class cannot be accessed via this reference without first casting it to the SuperClass class:

```
Object obj1 = iSuperTypeRef.method01(); // OK. method01 is in
InterfaceSuperType interface.
Object obj2 = iSuperTypeRef.method02();   // Not OK. method02 is
not in InterfaceSuperType interface.
Object obj3 = ((SuperClass) iSuperTypeRef).peek();     // OK.
Method in StackImpl class.
```

Polymorphism and Dynamic Method Lookup

Which object a reference will actually denote during runtime, cannot always be determined at compile time. Polymorphism allows a reference to denote objects of different types at different times during execution. A supertype reference exhibits polymorphic behavior, since it can denote objects of its subtypes.

When a non-private instance method is invoked on an object, the method definition actually executed is determined both by the type of the object at runtime and the method signature. Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the type of the object. However, a call to a private instance method is not polymorphic. Such a call can only occur within the class, and gets bound to the private method implementation at compile time.

Interfaces are constructs which do not contain any implementation. Interfaces only enforce a nature of behavior.
The real advantage offered by interfaces is that of poly-morphism. It is the ability to use an implementation via the interface's/supertype's reference. This offers real power because it offers the facility to plug and play with multiple versions of implementations.

Overloading on return values

It is common to wonder “Why only class names and method argument lists? Why not distinguish between methods based on their return values?” For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f() {}
int f() {}
```

This works fine when the compiler can unequivocally determine the meaning from the context, as in **int x = f( )**. However, you can call a method and ignore the return value; this is often referred to as *calling a method for its side effect* since you don’t care about the return value but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can Java determine which **f( )** should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

```
public abstract class AbstractClass
{
public AbstractClass()
{
      System.out.println("this is abstract classconstructor!");
}
public void aMethod()
{
      System.out.println("This method is in the abstract class");
}
}
```

No compiler error - the class cannot be instantiated directly. It has to be extended to an non-abstract class.
The constructors of the extended class will call the constructor of the abstract class (implicitly or explicitly).

- A collection allows a group of objects to be treated as a single unit
- The collections framework presents a set of standard utility classes for managing such collections
- Comprises of 3 parts
    - Core interfaces=>Collection,List,Set--SortedSet
    - Concrete Classes=>HashSet,TreeSet,ArrayList etc...
    - Static utility classes/Decorators

- For Implementation/Heirarchy ==>refer figure
- Collection or map only stores the reference to objects and not the actual objects.
- All the concrete classes implement the serializable and the cloneable interface Therefore, the object of these classes can be serialized and cloned.

Figure 11.2. The Core Collection Interfaces and Their Implementations

Figure 11.3. The Core Map Interfaces and Their Implementations

Collections

The Collection interface specifies the contract that all collections should implement. Some of the operations in the interface are optional, meaning that a collection may choose to provide a stub implementation of such an operation that throws an UnsupportedOperationException when invoked

The implementations of collections from the java.util package support all the optional operations in the Collection interface (see Figure 11.2 and Table 11.2).

Basic operations

```
int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(Object element)          // Optional
boolean remove(Object element)       // Optional
```

Bulk Operations

```
boolean containsAll(Collection c)
boolean addAll(Collection c)         // Optional
boolean removeAll(Collection c)      // Optional
boolean retainAll(Collection c)      // Optional
void clear()                         // Optional
```

Array operations

Convert collections to arrays

`Object[] toArray()`==returns an array with all the elements of collection

`Object[] toArray(Object a[])`==Stores all the elements of a collection in an array of specified type

1. **if length of array is > than no. of elements in an collection. remaining values are filled with null**
2. **If the array is too small, a new array of the same runtime type and appropriate size is created**
3. **If the runtime type of the specified array is not a supertype of the runtime type of every element in the collection, an `ArrayStoreException` is thrown.**

Eg.

```
public class exp
{
     Collection c=new ArrayList();

             public void add(){
                c.add("1");
                c.add("2");
                c.add("3");
                c.add(new Integer(2));//>>adding element of type
Integer
                c.add("4");
             }

        public void display(){
           System.out.println("c:::"+c);
        }

        public void array(){

           String []st=new String[2];
           String dd[]= (String[])c.toArray(st);//>>arraystore
exception
AS runtime type of the specified array of type String is not a
supertype of the runtime type of every element in the collection
as it contains integer element as well
          System.out.println("dd.length"+dd.length);
        }

        public static void main(String[] args)
         {
             exp obj = new exp();

            obj.add();
            obj.display();
            obj.array();
            obj.display();

         }

}
```

Iterators

All Known Subinterfaces: ListIterator
All Known Implementing Classes: BeanContextSupport.BCSIterator

- **Allows sequential access to the elements of a collection**.
- **obtained by calling the following method of the Collection interface:**

```
Iterator iterator()
```

Has following methods::

1. **boolean hasNext()==> Returns true if the underlying collection still has elements left to return**

2. **Object next()==>Moves the iterator to the next element in the underlying collection, and returns the current element. If there are no more elements left to return, it throws a NoSuchElementException.**

3. **Object remove()==>Removes the element that was returned by the last call to the next() method.**
   - **Invoking this method results in an IllegalStateException, if the next() method has not yet been called, or when the remove() method has already been called after the last call to the next() method.**

```
 eg.
    public void iterate()
    {
       Iterator it= c.iterator();
       Object o= it.next();
       it.remove();
       it.remove();//this will cause
IllegalStateException
    }
```

   - **This method is optional for an iterator, that is, it throws an UnsupportedOperationException if the remove operation is not supported.**

- The majority of the iterators provided in the java.util package are said to be fail-fast.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it

is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs. **src java docs**

- When an iterator has already been obtained, structurally modifying the underlying collection by other means will invalidate the iterator. Subsequent use of this iterator will throw a ConcurrentModificationException.

Eg.

```
public void iterate()
 {
        Iterator it= c.iterator();
        Object o= it.next();
        System.out.println("object"+o);
        c.remove("1"); //This will invalidate the iterator
        display();
        it.remove();//Using iterator here will result in
ConcurrentModificationException
 }
```

- The remove() method of an iterator is the only recommended way to delete elements from the underlying collection during traversal with an iterator.

The order in which the iterator will return the elements from an underlying collection depends on the traversal order supported by the collection

Sets

public interface Set extends Collection

- **Set interface do not allow duplicate elements.**
- **A set can contain at most one null value.**
- **Set interface does not define any new methods.**
- Duplicate elements must not replace old elements.

Bulk Operations and Set Logic	
Set Methods (a and b are sets)	**Corresponding Mathematical Operations**
a.containsAll(b)	**b < a (subset)**
a.addAll(b)	**a = a U b (union)**
a.removeAll(b)	**a = a – b (difference)**
a.retainAll(b)	**a = a n b (intersection)**
a.clear()	**a = Ø (empty set)**

HASHSET ==>Concrete class

```
java.lang.Object
└  java.util.AbstractCollection
└      java.util.AbstractSet
└          java.util.HashSet
```

All Implemented Interfaces: Cloneable, Collection, Serializable, Set
Direct Known Subclasses: JobStateReasons, LinkedHashSet

- **Uses hash Table**
- **A HashSet relies on the implementation of the hashCode () and equals() methods of its elements**
- **Unordered**

Constructors

- **HashSet()**

- **HashSet(Collection c) ==> Constructs a new set containing the elements in the specified collection. The new set will not contain any duplicates. This offers a convenient way to remove duplicates from a collection.**

- **HashSet(int initialCapacity)==> Constructs a new, empty set with the specified initial capacity.**

- **HashSet(int initialCapacity, float loadFactor)==> Constructs a new, empty set with the specified initial capacity and the specified load factor.**

LinkedHashSet==>Concrete Class

java.lang.Object
└ java.util.AbstractCollection
└ java.util.AbstractSet
└ java.util.HashSet
└ java.util.LinkedHashSet

All Implemented Interfaces: Cloneable, Collection, Serializable, Set

---

- **SubClass of HashSet**
- **LinkedHashSet guarantees that the iterator will access the elements in insertion order, that is, in the order in which they were inserted into the LinkedHashSet.**

Constructors ==> analogous to hashset

Lists

All Superinterfaces: Collection
All Known Implementing Classes: AbstractList, ArrayList, LinkedList, Vector

- **Lists are collections that maintain their elements in order**
- **Can contain duplicates.**
- **The elements in a list are ordered.**
- **The List interface also defines operations that work specifically on lists**

- **Object get(int index)==> Returns the element at the specified index.**

- **Object set(int index, Object element)// Optional ==> Replaces the element at the specified index with the specified element. It returns the previous element at the specified index.**

- **void add(int index, Object element) // Optional ==>Inserts the specified element at the specified index. If necessary, it shifts the element previously at this index and any subsequent elements one position toward the end of the list. *The inherited method add (Object) from the Collection interface will append the specified element to the end of the list.***

- **Object remove(int index) // Optional ==>Deletes and returns the element at the specified index, contracting the list accordingly. *The inherited method remove(Object) from the Collection interface will remove the first occurrence of the element from the list.***

- **boolean addAll(int index, Collection c) // Optional ==>Inserts the elements from the specified collection at the specified index, using the iterator of the specified collection. The method returns true if any elements were added.**

In a non-empty list, the first element is at index 0 and the last element is at size()-1. As might be expected, all methods throw an IndexOutOfBoundsException if an illegal index is specified.

// Element Search

- **int indexOf(Object o)**
- **int lastIndexOf(Object o)**

// Open Range-View

```
List subList(int fromIndex, int toIndex)
```

1. **This method returns a view of the list, which consists of the sublist of the elements from the index fromIndex to the index toIndex-1.**
2. **Any changes in the view are reflected in the underlying list, and vice versa.**

List Iterators

public interface ListIterator extends Iterator

```
interface ListIterator extends Iterator {

    boolean hasNext();
    boolean hasPrevious();

    Object next();          // Element after the cursor
    Object previous();      // Element before the cursor

    int nextIndex();        // Index of element after the cursor
    int previousIndex();    // Index of element before the cursor

    void remove();          // Optional
    void set(Object o);     // Optional
    void add(Object o);     // Optional
}
```

- **The ListIterator interface is a bidirectional iterator for lists.**
- **When traversing lists, it can be helpful to imagine a cursor moving forward or backward between the elements when calls are made to the next() and the previous() method, respectively.**
- **The element that the cursor passes over is returned.**
- **When the remove() method is called, the element last passed over is removed from the list.**
- **ListIterator listIterator()**
- **ListIterator listIterator(int index)==>starts traversing the list from the element indicated by the specified index.**

ArrayList==>Concrete class

```
java.lang.Object
└  java.util.AbstractCollection
└      java.util.AbstractList
└          java.util.ArrayList
```

All Implemented Interfaces: Cloneable, Collection, List, RandomAccess, Serializable

- **Resizable-array implementation of the List interface.**
- **Implements all optional list operations, and permits all elements, including null**

Constructors v

`ArrayList()`==> Constructs a new, empty ArrayList. An analogous constructor is provided by the LinkedList and Vector classes.

`ArrayList(Collection c)`==>Constructs a new ArrayList containing the elements in the specified collection. The new ArrayList will retain any duplicates. The ordering in the ArrayList will be determined by the traversal order of the iterator for the collection passed as argument. An analogous constructor is provided by the LinkedList and Vector classes.

`ArrayList(int initialCapacity)==>` Constructs a new, empty ArrayList with the specified initial capacity. An analogous constructor is provided by the Vector class.

LinkedList ==>concrete class

```
java.lang.Object
└  java.util.AbstractCollection
└      java.util.AbstractList
└          java.util.AbstractSequentialList
└              java.util.LinkedList
```

All Implemented Interfaces: Cloneable, Collection, List,

- **The LinkedList implementation uses a doubly-linked list.**
- **Insertions and deletions in a doubly-linked list are very efficient—elements are not shifted, as is the case for an array.**
- **The LinkedList class provides extra methods that**

implement operations that add, get, and remove elements at either end of a LinkedList:

```
void addFirst(Object obj)
void addLast(Object obj)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()
```

When frequent insertions and deletions occur inside a list, a LinkedList can be worth considering.

Vector
Simmilar to array >>>are synchronized

Position-based access has constant-time performance for the ArrayList and Vector classes. However, position-based access is in linear time for a LinkedList, owing to traversal in a doubly-linked list

Maps

- **A Map defines mappings from keys to values.**
- **The <key, value> pair is called an entry.**
- **A map does not allow duplicate keys.**

Basic operations

- **Object put(Object key, Object value) //optional**
- **Object get(Object key)**
- **Object remove(Object key) //optional**
- **boolean containsKey(Object key)**
- **boolean containsValue(Object value)**
- **boolean isEmpty()**
- **int size()**

Bulk operations

- void putAll(Map t) //optional
- void clear() //optional

Classes Implementing Map

- **HashMap**
- **LinkedHashMap,**
- **TreeMap**
- **Hashtable**

Class HashMap

java.lang.Object
└ java.util.AbstractMap
└ **java.util.HashMap**

All Implemented Interfaces: Cloneable, Map, Serializable
Direct Known Subclasses: LinkedHashMap, PrinterStateReasons

- **HashMap class is not thread-safe**
- **permits one null key**

Constructors

- **HashMap()**
- **HashMap(int initialCapacity)**
- **HashMap(int initialCapacity, float loadFactor)**
- **HashMap(Map otherMap)**

- **An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*.**
- **The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.**
- **The *loadfactor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased.**
- **When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method.**

Class LinkedHashMap

java.lang.Object
└ java.util.AbstractMap
└ java.util.HashMap
└ java.util.LinkedHashMap

All Implemented Interfaces: Cloneable, Map, Serializable

- **Hash table and linked list implementation of the Map interface, with predictable iteration order**
- **Use doubly-linked list**
- **By default, the entries of a LinkedHashMap are in key insertion order,**
- **LinkedHashMap can also maintain its elements in (element) access order, that is, the order in which its entries are accessed, from least-recently accessed to most-recently accessed entries.**
- **This ordering mode can be specified in one of the constructors of the LinkedHashMap class.**

Constructors

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)

The ordering mode is true for access order and false for key insertion order.

Interface Comparator==>>java.util

Precise control of ordering can be achieved by creating a customized comparator

```
int compare(Object o1, Object o2)
```

- **The compare() method returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object, according to the total order.**
- **Since this method tests for equality, it is strongly recommended that its implementation does not contradict the semantics of the equals() method**

Interface Comparable==>java.lang

All Known Implementing Classes: BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String, URI

`A class can define the natural order of its instances by implementing the Comparable interface`

class's compareTo method is referred to as its *natural comparison method*.

`int compareTo(Object o)`

- **This method returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, based on the natural order.**
- **It throws a ClassCastException if the reference value passed in the argument cannot be cast to the type of the current object.**

Objects implementing this interface can be used as

- elements in a sorted set
- keys in a sorted map
- elements in lists that are sorted manually using the Collections.sort() method

Interface SortedSet

All Superinterfaces: Collection, Set
All Known Implementing Classes: TreeSet

A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

All elements inserted into an sorted set must implement the Comparable interface (or be accepted by the specified Comparator).

Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) (or comparator.compare(e1, e2)) must not throw a ClassCastException for any elements e1 and e2 in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

Methods

- **SortedSet headSet(Object toElement)==>>** The headSet() method returns a view of a portion of this sorted set, whose elements are strictly less than the specified element

- **SortedSet tailSet(Object fromElement)==>>**the tailSet() method returns a view of the portion of this sorted set, whose elements are greater than or equal to the specified element

- **SortedSet subSet(Object fromElement, Object toElement)==>>** The subSet() method returns a view of the portion of this sorted set, whose elements range from fromElement, inclusive, to toElement, exclusive.

```
// First-last elements
```

- `Object first()=>>`returns the first element currently in this sorted set
- `Object last()==>>`returns the last element currently in this sorted set.
-

Both throw a NoSuchElementException if the sorted set is empty.

```
// Comparator access
```

`Comparator comparator()==>>` This method returns the comparator

associated with this sorted set, or null if it uses the natural ordering of its elements.

Interface SortedMap

All Superinterfaces: Map
All Known Implementing Classes: TreeMap

A Map that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

All elements inserted into an sorted map must implement the Comparable interface (or be accepted by the specified Comparator).

Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) (or comparator.compare(e1, e2)) must not throw a ClassCastException for any elements e1 and e2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

```
// Range-view operations
```

- **SortedMap headMap(Object toKey)**
- **SortedMap tailMap(Object fromKey)**
- **SortedMap subMap(Object fromKey, Object toKey)**

```
// First-last keys
```

- **Object firstKey()**
- **Object lastKey()**

```
// Comparator access
Comparator comparator()
```

Constructors

```
TreeSet()
TreeMap()
```

A standard constructor to create a new empty sorted set or map, according to the natural order of the elements or the keys, respectively.

```
TreeSet(Comparator c)
TreeMap(Comparator c)
```

A constructor that takes an explicit comparator for ordering the elements or the keys.

```
TreeSet(Collection c)
TreeMap(Map m)
```

A constructor that can create a sorted set or a sorted map based on a collection or a map, according to the natural order of the elements or the keys, respectively.

```
TreeSet(SortedSet s)
TreeMap(SortedMap m)
```

1. **Objects of a class that override the equals() method can be used as elements in a collection.**

2. **If they override the hashCode() method, they can also be used as elements in a HashSet and as keys in a HashMap.**

3. **Implementing the Comparable interface allows them to be used as elements in sorted collections and sorted maps**

An implementation of the equals() method must satisfy the properties of an equivalence relation:

- **Reflexive:For any reference self, self.equals(self) is always true.**
- **Symmetric:For any references x and y, x.equals(y) is true if and only if y.equals(x) is true.**
- **Transitive:For any references x, y and z, if both x.equals(y) and y. equals(z) are true, then x. equals(z) is true.**
- **Consistent:For any references x and y, multiple invocations of x. equals(y) always return the same result, provided the objects denoted by these references have not been modified to affect the equals comparison.**
- **null comparison:For any non-null reference obj, obj.equals(null) is always false.**

Example

```
public boolean equals(Object obj) {                 // (1)
        if (obj == this)                                // (2) Reflexive
            return true;
        if (!(obj instanceof UsableVNO))                // (3)
            return false;
        UsableVNO vno = (UsableVNO) obj;                // (4)
        return vno.patch    == this.patch    &&         // (5)
               vno.revision == this.revision &&
               vno.release  == this.release;
    }
```

Method overriding signature

The method prototype is

```
public boolean equals(Object obj)          // (1)
```

The signature of the method requires that the argument passed is of the type Object. The following header will overload the method, not override it:

```
public boolean equals(MyRefType obj)      // Overloaded.
```

The compiler will not complain. Calls to overloaded methods are resolved at compile time, depending on the type of the argument. Calls to overridden methods are resolved at runtime, depending on the type of the actual object denoted by the argument. Comparing the objects of the class MyRefType that overloads the equals() method for equivalence, can give inconsistent results:

```
MyRefType ref1 = new MyRefType();
MyRefType ref2 = new MyRefType();
Object    ref3 = ref2;
boolean b1 = ref1.equals(ref2);    // True. Calls equals() in MyRefType.
boolean b2 = ref1.equals(ref3);    // Always false. Calls equals() in Object.
```

Reflexivity test

Correct argument type

The equals() method should checks the type of the argument object at , using the instanceof operator:

```
if (!(obj instanceof UsableVNO))                // (3)
    return false;
```

This code also does the null comparison correctly, returning false if the argument obj has the value null.

The instanceof operator will also return true if the argument obj denotes a subclass object of the class UsableVNO. If the class is final, this issue does not arise—there are no subclass objects. **The test at (3) can also be replaced by the following code in order to exclude all other objects, including subclass objects:**

```
if ((obj == null) || (obj.getClass() != this.getClass()))   // (3a)
    return false;
```

The test in (3a) first performs the null comparison explicitly. The expression (obj.getClass() != this.getClass()) determines whether the classes of the two objects have the same runtime object representing them. If this is the case, the objects are instances of the same class.

Argument casting

The argument is only cast after checking that the cast will be successful. The instanceof operator ensures the validity of the cast, as done in Example 11.9. The argument is cast at (4) to allow for class-specific field comparisons:

```
UsableVNO vno = (UsableVNO) obj;                // (4)
```

Field comparisons

Equivalence comparison involves comparing certain fields from both objects to determine if their logical states match. For fields that are of primitive data types, their primitive values can be compared. Instances of the class UsableVNO in Example 11.9 have only fields of primitive data types. Values of corresponding fields are compared to test for equality between two UsableVNO objects:

```
return vno.patch    == this.patch    &&          // (5)
       vno.revision == this.revision &&
       vno.release  == this.release;
```

If all field comparisons evaluate to true, the equals() method returns true.

For fields that are references, the objects denotes by the references can be compared. For example,

if the UsableVNO class declares a field called productInfo, which is a reference, the following code could be used:

```
(vno.productInfo  == this.productInfo ||
(this.productInfo != null && this.productInfo.equals(vno.productInfo)))
```

The expression vno.productInfo == this.productInfo checks for the possibility that the two objects being compared have a common object denoted by both productInfo references. **In order to avoid a NullPointerException being thrown, the equals() method is not invoked if the this. productInfo reference is null.**

Exact comparison of floating-point values should not be done directly on the values, but on the integer values obtained from their bit patterns (see static methods Float.floatToIntBits() and Double.doubleToLongBits()). This technique eliminates certain anomalies in floating-point comparisons that involve a NAN value or a negative zero (see also the equals() method in Float and Double classes).

Only fields that have significance for the equivalence relation should be considered. Derived fields, whose computation is dependent on other field values in the object, might be redundant to include, or only including the derived fields might be prudent. Computing the equivalence relation should be deterministic, so the equals() method should not depend on unreliable resources, such as network access.

The order in which the comparisons are carried out can influence the performance of the equals comparison. Fields that are most likely to differ should be compared as early as possible in order to short-circuit the computation. In our example, patch numbers evolve faster than revision numbers, which, in turn, evolve faster than release numbers. This order is reflected in the return statement at (5) in Example 11.9.

Above all, an implementation of the equals() method must ensure that the equivalence relation is fulfilled.

General Contract of the hashCode() method

The general contract of the hashCode() method stipulates:

- **Consistency during execution:** Multiple invocations of the hashCode() method on an object must consistently return the same hash code during the execution of an application, provided the object is not modified to affect the result returned by the equals() method. The hash code need not remain consistent across different executions of the application. This means that using a pseudorandom number generator to produce hash values is not a valid strategy.

- **Object value equality implies hash value equality**: If two objects are equal according to the equals() method, then the hashCode() method must produce the same hash code for these objects. This tenet ties in with the general contract of the equals() method.

- **Object value inequality places no restrictions on the hash value:** If two objects are unequal according to the equals() method, then the hashCode() method need not produce distinct hash codes for these objects. It is strongly recommended that the hashCode() method produce unequal hash codes for unequal objects.

The collection implementations can be augmented with the following functionality:

- thread-safety
- collection immutability

```
static Collection synchronizedCollection(Collection c)
static List       synchronizedList(List list)
static Map        synchronizedMap(Map m)
static Set        synchronizedSet(Set s)
static SortedMap  synchronizedSortedMap(SortedMap m)
static SortedSet  synchronizedSortedSet(SortedSet s)
```

All threads must access the underlying collection through the synchronized view, otherwise, non-deterministic behavior may occur.

```
// Create a synchronized decorator.
Collection syncDecorator = Collections.synchronizedCollection
(nonsyncCollection);
```

In addition, for traversing a synchronized collection, the code for traversing the collection must be synchronized on the decorator:

```
// Each thread can only traverse when synchronized on
the decorator.

synchronized(syncDecorator) {
    for (Iterator iterator = syncDecorator.iterator();
iterator.hasNext();)
        doSomething(iterator.next());
}
```

```
static Collection unmodifiableCollection(Collection c)
static List       unmodifiableList(List list)
static Map        unmodifiableMap(Map m)
static Set        unmodifiableSet(Set s)
static SortedMap  unmodifiableSortedMap(SortedMap m)
static SortedSet  unmodifiableSortedSet(SortedSet s)
```

The unmodifiable views intercept all calls that can modify the underlying collection, and throw an UnsupportedOperationException.

If the view is a collection, then this restriction applies for any iterator of this collection.

In the case of a map, the restriction also applies to any collections created from this map.

??/

```
static void sort(List list)

static void sort(List list, Comparator comp)
```

The first method sorts the elements in the list according to their natural order.

The second method does the sorting according to the total ordering specified by the comparator

- `static int binarySearch(List sortedList, Object obj)=>>`The methods use a binary search to find the index of the obj element in the specified sorted list.
- `static int binarySearch(List sortedList, Object obj, Comparator comp)>>R` equires that it is sorted according to the total ordering dictated by the comparator.

an immutable collection or map containing only one element or one entry, respectively can be created by calling the following static factory methods of the Collections class, respectively:

```
static Set  singleton(Object o)
static List singletonList(Object o)
static Map  singletonMap(Object key, Object value)
```

static void copy(List dst, List src)
Adds the elements from the src list to the dst list.

static void fill(List list, Object o)
Replaces all of the elements of the list with the specified element.

static List nCopies(int n, Object o)
Creates an immutable list with n copies of the specified object.

static void reverse(List list)
Reverses the order of the elements in the list.

static Comparator reverseOrder()
Returns a comparator that enforces the reverse of the natural ordering. Useful for maintaining objects in reverse-natural order in sorted collections and arrays.

The following code conjures up a modifiable list initialized with 99 null elements:

```
List itemsList = new ArrayList(Collections.nCopies(99, null));
```

This code would sort a list of Integers and an array of strings in descending order and in inverse-lexicographical order, respectively:

```
Collections.sort(intList, Collections.reverseOrder());
Arrays.sort(strArray, Collections.reverseOrder());
```

The elements in the following set would be maintained sorted in descending order:

```
Collection intSet = new TreeSet(Collections.reverseOrder());
intSet.add(new Integer(9));  intSet.add(new Integer(11));
intSet.add(new Integer(-4)); intSet.add(new Integer(1));
System.out.println(intSet);          // [11, 9, 1, -4]
```

```
static void shuffle(List list)
```

Randomly permutes the list, that is, shuffles the elements.

```
boolean replaceAll(List list, Object oldVal, Object newVal)
```

Replaces all elements equal to oldVal with newVal in the list; returns true if the list was modified.

```
static void rotate(List list, int distance)
```

Rotates the elements towards the end of the list by the specified distance. A negative value will rotate toward the start of the list.

```
static void swap(List list, int i, int j)
```

Swaps the elements at indices i and j.

The effect of these utility methods can be limited to a sublist, that is, a segment of the list. The following code illustrates rotation of elements in a list. Note how the rotation in the sublist view is reflected in the original list.

```
// intList denotes the following list:                          [9, 11, -4, 1,
7]
Collections.rotate(intList, 2);        // Two to the right.     [1, 7, 9, 11, -
4]
Collections.rotate(intList, -2);       // Two to the left.      [9, 11, -4, 1,
7]
List intSublist = intList.subList(1,4);// Sublist:                 [11, -4, 1]
Collections.rotate(intSublist, -1);    // One to the left.        [-4, 1, 11]
                                       // intList is now:       [9, -4, 1, 11,
7]
```

Utility Methods in the Arrays Class

The Arrays class provides useful utility methods that operate on arrays: binary search, sorting, array comparison, array filling.

The Arrays class also provides the static asList() method, which can be used to create List views of arrays. Changes to the List view reflect in the array, and vice versa. The List is said to be backed by the array. The List size is equal to the array length and cannot be changed. The asList() method in the Arrays class and the toArray() method in the Collection interface provide the bidirectional bridge between arrays and collections.

```
static List asList(Object[] backingArray)
```

```
String[] jiveArray       = new String[] {"java", "jive", "java", "jive"};
Set      jiveSet         = new HashSet(Arrays.asList(jivearray));        // (1)
String[] uniqueJiveArray = (String[]) jiveSet.toArray(new String[0]);    // (2)
```

At (1), the jiveArray is used to create a List, which, in turn, is used to create a Set. At (2) the argument to the toArray() method specifies the type of the array to be created from the set. The final array uniqueJiveArray does not contain duplicates.

Concrete Collection/ Map	Interface	Duplicates	Ordered/ Sorted	Methods Called on Elements	Data Structures on Which Implementation Is Based
HashSet	Set	Unique elements	No order	equals() hashCode()	Hash table
LinkedHashSet	Set	Unique elements	Insertion order	equals() hashCode()	Hash table and doubly-linked list
TreeSet	SortedSet	Unique elements	Sorted	equals() compareTo()	Balanced tree
ArrayList	List	Allowed	Insertion order	equals()	Resizable array
LinkedList	List	Allowed	Insertion order	equals()	Linked list
Vector	List	Allowed	Insertion order	equals()	Resizable array
HashMap	Map	Unique keys	No order	equals() hashCode()	Hash table
LinkedHashMap	Map	Unique keys	Key insertion order/Access order of entries	equals() hashCode()	Hash table and doubly-linked list
Hashtable	Map	Unique keys	No order	equals() hashCode()	Hash table
TreeMap	SortedMap	Unique keys	Sorted in key order	equals() compareTo()	Balanced tree

Set

implementations of the Set interface do not allow duplicate elements.⇒can contain at most one null value.
The Set interface does not define any new methods, and its add() and addAll() methods will not store duplicates.
üÜIf an element is not currently in the set, two consecutive calls to the add() will first return true, then false.

HashSet

Since this implementation uses a hash table, it offers near constant-time performance for most operations.
üÜdoes not guarantee any ordering of the elements.
the LinkedHashSet subclass of HashSet guarantees insertion-order, i.e., the iterator will access the elements in insertion order.
Sorted counterpart is TreeSet, which implements the SortedSet interface and has logarithmic time complexity.
A HashSet relies on the implementation of the hashCode() and equals() methods of its elements. The hashCode() is used for hashing the elements, and the equals() method is needed for comparing elements.

```
HashSet()
HashSet(Collection c)
HashSet(int initialCapacity)
HashSet(int initialCapacity, float loadFactor)
```

By default InitialCapacity is 16 and loadFactor is 0.75 (3/4). The Capacity of Map is doubled (capacity<<2) each time 3/4th(Load Factor) of the capacity is full. This limit is called threshold (= capacity*loadFactor) ==> By Default, Threshold is 12.

HashSet Uses HashMap internally.

```
private transient HashMap map;
private static final Object PRESENT = new Object();

public HashSet()
{
    map = new HashMap();
}

public boolean add(Object obj)
{
    return map.put(obj, PRESENT) == null;
}

public boolean remove(Object obj)
{
    return map.remove(obj) == PRESENT;
}
```

TreeSet

TreeSet automatically sorts an element into its correct place in the collection whenever you add it!
The sort order will either be the natural order for the class, as expressed by Comparable,

or the order defined by a Comparator that you pass to the constructor. Iterator for a TreeSet guarantees to deliver the elements in this order.

A TreeSet collection is implemented by ("has a") a TreeMap behind the scenes. TreeMap in turn uses a red/black tree (Binary Balanced Tree) as its data structure.

Map

A Map defines mappings from keys to values. The <key, value> pair is called an entry. A map does not allow duplicate keys.
Both the keys and the values must be objects.
The Map can be traversed using different collection views: a key set, a value collection, or an entry set.

Basic Operations:

Object put(Object key, Object value) >> [Optional] Inserts the <key, value> entry into the map.
It returns the value previously associated with the specified key, if any. Otherwise, it returns the null value.

Object get(Object key) >> Returns the value mapped the key or null if no entry is found.

Object remove(Object key) [Optional] >> Returns the value mapped with the key, if any. Otherwise, null.

boolean containsKey(Object key)
boolean containsValue(Object value)
int size() >> Number of Entries
boolean isEmpty()

Bulk Operation

void putAll(Map t) >>Optional
void clear() >>Optional

Collection View

Changes in the map are reflected in the view, and vice versa.

Set keySet() >> Retruns an object of HashMap$KeySet Class >> AbstractSet
Collection values() >> Returns an object of HashMap$Values Class >> AbstractCollection
Set entrySet() >>...HashMap$EntrySet >>...

All these Objects keeps an object of HashMap internally to work on. Thats y, any change made is visible in the base HashMap.

An Effort to add() in any of the View, will throw an UnsupportedOperationException << Behaviour defined in Super Class AbstractCollection

Each <key, value> entry is represented by an object implementing the nested Map.Entry interface.
interface Entry {

```
    Object getKey();
    Object getValue();
    Object setValue(Object value);
}
```

Implementations of Map

HashMap >> Unordered Map >>1 Null Key
LinkedHashMap >> Ordered Map
TreeMap >> Sorted Map
Hashtable >> Unordered Map >>Thread Safe >> Non-Null Keys/ Value

LinkedHashMap >> Order Dosen't change on reinserting a key <<< As no new entry is created.
By default, the entries of a LinkedHashMap are in the order in which the keys are inserted in the map (insertion order). This ordering mode can be specified in one of the constructors of the LinkedHashMap class. e.g., can arrange its elements in access order, i.e., the order in which its entries are accessed, from least-recently accessed to most-recently accessed entries.

Implementation of an Entry of LinkedHashMap

```
class LinkedHashMap$Entry extends HashMap.Entry{
        :
        :
    LinkedHashMap$Entry before; >> Previous Entry
    LinkedHashMap$Entry after;    >> Next Entry
}
```

Adding, removing, and finding entries in a LinkedHashMap can be slightly slower than in a HashMap, as an ordered doubly-linked list has to be maintained.
Traversal of a map is through one of its collection-views.
For an underlying LinkedHashMap, the traversal time is proportional to the size of the map—regardless of its capacity. However, for an underlying HashMap, it is proportional to the capacity of the map.

????
In case of a HashMap, to find next entry in table[], it is required to traverse thru the table until it encounters a non-null entry. This made it traverse till the end of the table[], i.e., capacity of the map.

```
        HashMap.Entry aentry[] = table;
        for(int i = index; entry1 == null && i > 0; entry1 = aentry[--i]);    <<travarse untill a non-null entry is encountered
        index = i;          << cursor position in the table
        next = entry1;
        return current = entry;<< Validate the Current Pointer
```

In case of a LinkedHashMap, each entry keeps a reference to the next entry in the Map, i.e., it just has to traverse for the size of the Map.

```
          nextEntry = entry.after;
```

The concrete map classes override the toString() method. The standard textual

representation generated by the toString() method for a map is

{key1=value1, key2=value2, ..., keyn=valuen}

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
HashMap(Map m)
```

Serialization of MAP and SET??? <<< Because Entry Table etc are transient

Some of the operations in the interface are optional, meaning that a collection may choose to provide a stub implementation of such an operation that throws an UnsupportedOperationException when invoked.

Basic Operations

returns boolean
add(Object) >> is add successful (false when a duplicate value is provided to add in a set)
remove(Object) >> element Not found
contains(Object)
isEmpty()
return int
size()

Bulk Operations

returns boolean
addAll(Collection) >>if any change is made in the collection
removeAll(Collection) >> " >> Minus
retainAll(Collection) >> " >> Intersection
containsAll(Collection) >>subset
returns void
clear() >> removes all the elements [Optional]

Array Operations

Object[] toArray()
Object[] toArray(Object a[])

- assigns the element in the Collection to the Provided array
  - if(element Types are not compatible) >>>ArrayStoreException
  - if(a.length < this.size()) >>>Create a new Array of the element type and returns the same
  - if(a.length > this.size()) >>>unoccupied indexes will be filled with null values
  - if(a == null) >>> illegalArgumentException

Interface Iterator

each collection provides an iterator to facilitate sequential access of its elements.

boolean hasNext()
Object next() >>> NoSuchElementException when no element is left
Object remove() >>> [Optional]
Removes the current element. (when a remove() is called, it invalidates the current pointer (-1/null) >>> any further call of the remove() method will throw an IllegalStateException) when next() is called, the current pointer is validated (idx+1/ currentElem)
Implementation:
int expectedModCount >>> its value is checked with the modCount in the corresponding collection whenever it is accessed thru an iterator >>> if this check fails >>> a ConcurrentModificationException is thrown

```
final void checkForComodification()
{
    if(modCount != expectedModCount)
        throw new ConcurrentModificationException();
```

```
        else
            return;
    }
```

this expectedModCount is decremented when a remove() is called on the iterator.

Iterator - Indentification of concurrent modification modCount

`protected transient int` **`modCount`** The number of times this list has been *structurally modified*.
Its a field in each collection class.
Structural modifications are those that change the size of the list, or otherwise upset (change) it in such a fashion that iterations in progress may yield incorrect results.

This field is used by the iterator and list iterator implementation returned by the iterator and listIterator methods.
If the value of this field changes unexpectedly, the iterator (or list iterator) will throw a ConcurrentModificationException in response to the next, remove, previous, set or add operations.
This provides *fail-fast* behavior, rather than non-deterministic behavior in the face of concurrent modification during iteration.

Use of this field by subclasses is optional.
If a subclass wishes to provide fail-fast iterators then it has to increment modCount field in its add(int, Object) and remove(int) methods (and any other methods that result in structural modifications to the list).
If an implementation does not wish to provide fail-fast iterators, this field may be ignored.

expectedModCount

this is a field in the iterator. Thats used to check a concurrent structural modification

```
    final void checkForComodification()
    {
        if(modCount != expectedModCount)
            throw new ConcurrentModificationException();
        else
            return;
    }
```

fail-fast iterators ==> no concurrent Structural modification to the corresponding collection class. If identified such changes then a ConcurrentModificationException is thrown on next access of the collection thru iterator.

List subList(int fromIndex, int toIndex)

returns an Object of Sublist class that extends AbstractList class.

```
class SubList extends AbstractList{
    private AbstractList l;
    private int offset; <<<<<<<Start
    private int size;  <<<
    private int expectedModCount;
          :

<<<Default>>>SubList(AbstractList abstractlist, int i, int j) << No access from outside the package
{
          l = abstractlist;
```

```
        offset = i;
        size = j - i;
        expectedModCount = l.modCount;
}
public List subList(int i, int j){..}
```

>>>> any Changes to the structure of the Sublist is also reflected in the base List >>>>
Collections.sort(subList), Collections.reverse(subList)...too

Initializer is called before constructor call -->after the call to super()

Polymorphism is applicable in case of METHODS only NOT data members (Static & Non-Static)

```
class Cls1{
        public  int staticVar ;
        public Cls1(){ }
}
class Cls2 extends Cls1{
        {
                System.out.println("initializer...");
        }

        public int staticVar = 21;
        public Cls2(){
                System.out.println("static var: "+ this.staticVar);
        }
}
class Main{
        public static void main(String arg[]){
                Cls1 obj = new Cls2();
                System.out.println("obj.staticVar:  "+obj.staticVar);  ///<<< Output will be 0 << the instance
var frm base class
        }
}
```

Overriding:
Access Specifier: Widening is possible But, Narrowing isn't possible

Method definition in the subclass MUST have same signature and the same RETURN type. << NOT compatible==>> SAME

function that Throws Exception: --> Mentioning Checked Exception matters.. mentioning/not unchecked Exceptions won't effect..

if overridden the the signature must throw a compatible exceptions (Same/Subclass of that) (one or more than one).
super class
        throws IOException
sub-class
        throws IOException, ClassCastException << unchecked Exception

A mention of unchecked expection in the subclass overridden function doesn't result in violation.

sub-class throws IOException, InterruptedException << both checked exception
InterruptedException isn't compatible to the IOException

Polymorphism:
compile time --> compiler checks for whether the method is accessable in the reference.
obj1 = new Cls2();
obj1.method4();

even if method4 is present in Cls2 --> the compile time error will result

++> private method2() in Cls1
and public method2() in Cls2   [widenning]

obj1 = new Cls2();
obj1.method2(); ==> results in compile time error

Object Initialization:
Compiler creates an implicit initializer that initializes all the data members to their default/specified value. ==> Cls1 obj = new Cls1(); in the Class C==> will result in call of the Constructor of Cls1 B4 the call of C class const.. any SOP in Cls1() will be shown B4 the one in the C1()

On Object Initialization,

```
class Cls1{
	public static int staticVar ;
	public Cls1(){System.out.println("CLS!"); }

	public String method1(int i){
		System.out.println("Cls1.method1");
		return "1";
	}
}



class Cls2 extends Cls1{
	{
		System.out.println("initializer... Cls2");
	}

	public int staticVar = 21;
	public static Cls3 obj3 = new Cls3();	<< Result in a implicit static Initializer (called at class load)
	public Cls1 obj = new Cls1();

	public Cls2(){
		System.out.println("static var: "+ this.staticVar);
	}

	public String method1(int i){
		System.out.println("Cls2.method1");
		return "2";
	}
}

class Cls3{
	public Cls3(){
		System.out.println("Cls3");
	}
}

class Main{
	public static void main(String arg[]){
		Cls2 obj = new Cls2();
		System.out.println("obj.method1:  "+obj.method1(1));
	}

	public Main(){
		System.out.println();
	}
}

=====================================================
Cls3			<< Implicit Static Initializer
CLS!			<< Constructor Call to superClass	<< Implicit Call to super() [default
consturctor]
initializer... Cls2	<< explicit Initializer
CLS!			<< Implicit Initializer <<]
static var: 21		<< Constructor Call
Cls2.method1
obj.method1:  2
=====================================================

super(i)
```

Initializer Expression >> compiler creates Initializer block entry for each

static field initialization --> implicit static initializer
non-static field initialization --> implicit initializer

Static initializer is called at the time of class load. => b4 non-static field initialization

An instance initializer expression can always refer to any static member of the class, regardless of the member declaration order. B'cz static ones have already been initialized

Forward Referencing: [Declaration B4 read Rule]
==========
```
class..{
//int field1 = field2*5; <<<<no forward referencing
int field1 = this.field2*5;  //alowed
int field2 = 1;
}
```

```
class Cls2 extends Cls1{
        {
                System.out.println("initializer... Cls2");
        }
        //int k = 4*j;   <<<<illegal forward reference
        int i = 5*this.j;  // temp. initialization to the default value 0  >>>> no entry is made into symbol table
        //int k = 4*j;   <<<<illegal forward reference                          >> thts y not found j
        int j = 10;
```

Fields:
Declaration is must b4 read of the field.

Forward Referencing:
writing can be possible b4 Declaration.
```
int i =j =10;
int j;
```

The rule is a variable should be declared b4 its value is read, bUT it can be written b4 its declaration.
int i = j =10; >>>>> int i =(j=10); initializing with 10 not with the value of j; <<<<<<

But, int i =j*10; >> not allowed fwd referencing

```
:
   int k=called();  // method uses l >>> using l b4 declaration
        int j =10;
        int l=8;
:
:
        private int called(){
                System.out.println("this.l: "+l); // will output wat this.l with do when RB4I //<< function
accesses the instance fields using this.
// output is gonna be 0 as this.l is accessed b4 initialization >>> k = 0
                return l;
        }
```
---

if an initialization can throw a checked exception, that must be caught. >>>compilation error

```
int k=called();
```

```
private int called(){
                System.out.println("this.l: "+l);
                throw new IOException();
                return l;
}
```

Execution of the initializer statements and the initializer block in the class is in the same order as they appear in the class.

```
        int k=called();
            int j =10;
            {
                System.out.println("k: "+k+" j: "+j+" l: "+this.l); //this.l will be 0 as forward referencing
                l=3;
                System.out.println("k: "+k+" j: "+j+" l: "+this.l);// this.l=3
            }
            int l=8;
            {
                System.out.println("k: "+k+" j: "+j+" l: "+this.l);

            }


_Output:
k: 0 j: 10 l: 0
k: 0 j: 10 l: 3
k: 0 j: 10 l: 8
```

Same rules for static initialization:
static don't have this so can't be fwd reference at RB4I

Static field cant be initialized to not static

---

try{}catch() in block is allowed in the initializer
to catch a thrown exception

In case of instance initializer,
try & catch can be avoided if the a compatible exception is thrown in every constructor of class.

=> The instance creation should be enclosed in the try{}catch(){}/functions throws an exception

```
            public Main(Cls5 obj5)  throws Exception{
                obj5.super();
                System.out.println("cOnstructor:  Main with obj5");
            }

            public static void main(String arg[]) throws Exception{
                Main mn = new Main(obj5);// this should be in try{}catch(){}... the function must throw the
exception
            }

            int k= called();  //if all the constructors of the class don't throw the checked exception, compilation
error
            int j =10;

            {
                System.out.println("k: "+k+" j: "+j+" l: "+this.l);
                l=3;
```

```
                System.out.println("k: "+k+" j: "+j+" l: "+this.l);

                if(true)
                        throw new Exception();  //if all the constructors of the class don't throw the checked exception, compilation error
        }

        private  static int called() throws IOException{
                System.out.println("this.l: "+1);
                if(true)
                throw new IOException();
                return 1;
        }
```

```
class Main extends Cls5.Cls5_1{
	Cls1 obj5 = new Cls1(1);

	public static void main(String arg[]) throws Exception{
		Cls5 obj6 = new Cls5(3);
		Main mn = new Main(obj6);
		mn.callIt();
	}

	public void callIt() throws Exception{
		obj5 = null; // eligible for garbage collection
		System.gc();
		while(true){
			System.out.println("abhi jinda hOon mein");
			Thread.currentThread().sleep(1000);
		}
	}
}
```

???? Is this tru????
If object obj1can access object obj2 that is eligible for garbage collection, then obj1 is also eligible for garbage collection.

If obj1 isn't don't any kinda lafda like making it null wagara wagara...
then if obj2 is eligible for GC then obj1 will also be...

******* Circular References don't prevent objects from being garbage collected??? only reachable references will do...**
If object obj1 accessible from object obj2 and obj2 is accessible from obj1 then obj1 and obj2 are elligible for garbage collection

```
class Cls6{
	public Cls5 obj5 = new Cls5(1);
	protected void finalize(){
		System.out.println("Finalize of Cls6...");
	}
}

class Cls2 extends Cls1{
	public Cls5 obj5;
	public void setCls5(Cls5 obj){
		obj5 = obj;
	}
	protected void finalize(){
			System.out.println("Finalize of Cls2...");
	}

}

class Cls5{
	public Cls2 getCls2(){
		return obj2;
	}

	public Cls2 obj2= new Cls2(2);

	protected void finalize(){
			System.out.println("Finalize of Cls5...");
	}
}


Main<<<<

			public void callIt() throws Exception{

				Cls6 obj6 = new Cls6();
				Cls5 obj5 = obj6.obj5;
				Cls2 obj2 = obj5.getCls2();

				obj2.setCls5(obj5);

				obj5 = null;
				obj2 = null;
				obj6 = null; // as there isn't any reference of obj2 and obj5 on stack these
are eligible for garbage collection
						// and there isn't any other object reference accessing
these object
						// if obj6 isn't made null then these aren't elligible for
garbage collection.. as there is obj6 on 		// stack with
access to obj2 and obj5
				System.gc();
				while(true){
					System.out.println("abhi jinda hOon mein");
					Thread.currentThread().sleep(1000);
			}


Output:
```

```
::
:
:
Finalize of Cls6...
Finalize of Cls5...
Finalize of Cls2...
Cls1 Finalize
```

Conclusion: as long as there is any reachable reference to an object directly or indirectly, not elligble for garbage collection.

reachable referrence << reference on any stack...

>> if obj6 = null; is commented the obj2 and obj5 aren't elligble for GC<< as indirect reachable reference thru obj6(present in stack)

1. objects is allocated in a designated part of memory called the heap.
2. Garbage collection is a process of managing the heap efficiently; that is, reclaiming memory occupied by objects that are no longer needed and making it available for new objects
3. The automatic garbage collector figures out which objects are not reachable and, therefore, eligible for garbage collection
4. It will certainly go to work if there is a danger of running out of memory.
5. A program has no guarantees that the automatic garbage collector will be run during its execution. A program should not rely on the scheduling of the automatic garbage collector for its behavior
6. An automatic garbage collector essentially performs two tasks:
    a. decide if and when memory needs to be reclaimed
    b. find objects that are no longer needed by the program and reclaim their storage
7. Local references declared in a method can always be found in the **method's activation record**, on the runtime **stack** associated with the thread in which the method is called
8. **Objects**, on the other hand, are always created in the **heap**.
    a. If an object has a field reference, then the field is to be found inside the object in the heap (reference is not on the stack),
    b. and the object denoted by the field reference is also to be found in the heap.
9. **An object in the heap is said to be reachable [=accessible]**
    a. **if it is referenced by any local reference in a runtime stack.**
    b. **Additionally, any object that is referenced by a reference within a reachable object.**
10. **Reachability is a transitive relation.**
    a. **Thus, a reachable object has at least one chain of reachable references from the runtime stack.**
    b. ***Any reference that makes an object reachable is called a reachable reference*. An object that is not reachable is said to be unreachable.**
11. **A reachable object is alive**. It is accessible by the live thread that owns the runtime stack
12. Any object that is not accessible by a live thread is a candidate for garbage collection.
13. An object is eligible for garbage collection if all references denoting it are in eligible objects.
14. if a composite object becomes unreachable, then its constituent objects also become unreachable, barring any reachable references to the constituent objects
15. ******* Circular References don't prevent objects from being garbage collected??? only reachable references will do...**
16. If object obj1 accessible from object obj2 and obj2 is accessible from obj1 then obj1 and obj2 are elligible for garbage collection
17. if object obj1can access object obj2 that is eligible for garbage collection, then obj1 is also eligible for garbage collection.

18. if an object `obj2` is eligible for garbage collection and object `obj1` contains a reference to it, then object obj1 must also be eligible for garbage collection.

19. The abstract class `java.lang.ref.Reference` and its concrete subclasses (`SoftReference`, `WeakReference`, `PhantomReference`) provide **reference objects** that can be used to maintain more sophisticated kinds of references to another object (called the **referent**). A reference object introduces an extra level of indirection, so that the program does not access the referent directly. The automatic garbage

collector knows about reference objects and can reclaim the referent if it is only reachable through reference objects. The concrete subclasses implement references of various strength and reachability, which the garbage collector takes into consideration.

20. **Objects that are created and accessed by local references in a method are eligible for garbage collection when the method terminates, unless reference values to these objects are exported(**is returned from the method, passed as argument to another method that records the reference, or thrown as an exception**) out of the method**

21. The automatic garbage collector calls the **`finalize()`** method in an object that is eligible for garbage collection before actually destroying the object.
22. The **`finalize()`** method is defined in the `Object` class.

```
protected void finalize() throws Throwable
```

23. An implementation of the `finalize()` method is called a finalizer.
24. **A finalizer can, like any other method, catch and throw exceptions**
25. However, any exception thrown but not caught by a finalizer invoked by the garbage collector is ignored.
26. When the garbage collector calls the `finalize()` method, it will ignore any exceptions thrown by the `finalize()` method. In all other cases, normal exception handling occurs when an exception is thrown during the execution of the `finalize()` method**, that is, exceptions are not simply ignored.**
27. **The finalizer is only called once on an object.**
28. In case of finalization failure, the object still remains eligible for disposal at the discretion of the garbage collector (unless it has been resurrected,).
29. Overloading the `finalize()` method name is allowed, but only the method with the original signature will be called by the garbage collector.
30. The `finalize()` method in `Object` is `protected`. This means that overriding methods must be declared either `protected` or `public`.
31. **The `finalize()` method in `Object` can throw any `Throwable` object**.
32. **Overriding methods can limit the range of throwables to unchecked exceptions. <u>Further overridden definitions of this method in subclasses will not be able to throw checked exceptions.</u>**
33. Since there is no guarantee that the garbage collector will ever run, there is also no guarantee that the finalizer will ever be called.

```
  public class AnotherWellbehavedClass {
  SomeResource objRef;
  // ...
  protected void finalize() throws Throwable {          // (1)
   try {                                                // (2)
       if (objRef != null) objRef.close();
   } finally {                                          // (3)
       super.finalize(); <<<<<<mandatory   but will not cause compile time
error                          // (4)
   }
 }
}
```

34. A finalizer may make the object accessible again (i.e., resurrect it),
35. One simple technique is to assign its `this` reference to a static field, from which it can later be retrieved.
36. Since a finalizer is called only once on an object before being garbage collected, an object can only be resurrected once
37. an explicit call to the finalizer() method is possible within a method. bUt, It won't gc the object.

38. The **`System.gc()`** method can be used to request garbage collection,
39. **`System.runFinalization()`** method can be called to suggest that any pending finalizers be run for objects eligible for garbage collection.
40. The `Runtime` class provides various methods related to memory issues.
    `static Runtime getRuntime()``Returns the Runtime object associated with the current application.`
    `void gc()`

void runFinalization():::: Requests that any pending finalizers be run for objects eligible for garbage collection. Again

41. **Long freeMemory()>>>>>**Returns the amount of free memory (bytes) in the JVM, that is available for new objects.
    `long totalMemory()>>>>>>>``Returns the total amount of memory (bytes) available in the JVM. This includes both memory occupied by current objects and that which is available for new objects.`

1. Initializers can be employed for initialization of fields in objects and classes,
    a. field initializer expressions
    b. static initializer blocks
    c. instance initializer blocks
2. An initializer expression for a static field cannot refer to non-static members by their simple names.
3. The keywords **this and super** cannot occur in a static initializer expression.
4. Static initializer is called at the time of class load
5. An instance initializer expression can always refer to any static member of the class, regardless of the member declaration order. B'cz static ones have already been initialized.
6. Forward Referencing: [Declaration B4 read Rule]
    a. The rule is a variable should be declared b4 its value is read, bUT it can be written b4 its declaration.
7.

```
class..{
//int field1 = field2*5; <<<<no forward referencing
int field1 = this.field2*5;  //alowed
int field2 = 1;
}
```

```
class Cls2 extends Cls1{
        {
                System.out.println("initializer... Cls2");
        }
        //int k = 4*j;   <<<<illegal forward reference
        int i = 5*this.j;  // temp. initialization to the default value 0  >>>> no entry is made into symbol table
        //int k = 4*j;   <<<<illegal forward reference                    >> thts y not found j
        int j = 10;
```

Fields:
Declaration is must b4 read of the field.

Forward Referencing:

writing can be possible b4 Declaration.
int i =j =10;
int j;

The rule is a variable should be declared b4 its value is read, bUT it can be written b4 its declaration.
int i = j =10; >>>>> int i =(j=10); initializing with 10 not with the value of j; <<<<<<

But, int i =j*10; >> not allowed fwd referencing

```
:
   int k=called();  // method uses l >>> using l b4 declaration
        int j =10;
        int l=8;
:
:
        private int called(){
                System.out.println("this.l: "+l); // will output wat this.l with do when RB4I //<<
function accesses the instance fields using this.
// output is gonna be 0 as this.l is accessed b4 initialization >>> k = 0
                return l;
        }
```

---

if an initialization can throw a checked exception, that must be caught.
>>>compilation error

```
int k=called();

private int called(){
                System.out.println("this.l: "+l);
                throw new IOException();
                return l;
}
```

---

8. **if an initialization can throw a checked exception, that must be caught.**

   >>>**compilation error** int k=called(); // if **<u>all the constructors</u>** don't catch this exception

```
   private int called(){
                System.out.println("this.l: "+l);
                throw new IOException();
                return l;
}
```

9. **If any checked exception is thrown during execution of an initializer expression, it must be caught**.

10. **This restriction does not apply to <u>instance initializer expressions in anonymous classes.</u>**

    a. **But the enclosing context should handle it.**

b. Public void ss() **throws Exception**{ //handle exp thrown
Thread t= new Thread(){
int c;
{ if(c==0) Throw new Exception(); }
public void run(){}}; …}

11. Same rules for static initialization.
    a. static don't have this so can't be fwd reference at RB4I
12. Static field cant be initialized to not static

---

try{}catch() in block is allowed in the initializer

13. If a class relies on native method implementations, a static initializer can be used to load any external libraries that the class needs
14. A class can have more than one static initializer block. Initializer blocks are not members of a class nor can they have a return statement
15. **The keywords this and super cannot occur in a static initializer block.**
16. **I**

```
class StaticForwardReferences {

   static {            // (1) Static initializer block
      sf1 = 10;        // (2) OK. Assignment to sf1 allowed
   //  sf1 = if1;       // (3) Not OK. Non-static field access in static context
   //  int a = 2 * sf1;  // (4) Not OK. Read operation before declaration
      int b = sf1 = 20; // (5) OK. Assignment to sf1 allowed
      int c = StaticForwardReferences.sf1;// (6) OK. Not accessed by simple name
 <<<<<similar to this.field name access
   }

   static int sf1 = sf2 = 30;  // (7) Static field. Assignment to sf2 allowed
   static int sf2;             // (8) Static field
   int if1 = 5;                // (9) Non-static field

   static {            // (10) Static initializer block
      int d = 2 * sf1;  // (11) OK. Read operation after declaration
      int e = sf1 = 50; // (12)
   }

   public static void main(String[] args) {
      System.out.println("sf1: " + StaticForwardReferences.sf1);
      System.out.println("sf2: " + StaticForwardReferences.sf2);
   }
}
```

17. **Exception handling in instance initializer blocks differs,**

a. **the execution of an instance initializer block can result in an uncaught checked exception, provided the exception is declared in the throws clause of every constructor in the class.**

18. **Static initializer blocks cannot allow this, since no constructors are involved in class initialization.**

19. **Instance initializer blocks in anonymous classes have greater freedom: they can throw any exception**.

    a. **PROVIDED the enclosing context should handle the thrown exception**

20. **static {;}<<<<<<** empty initializers are valid

21. *****

    public class MyClass {

static int gap = 10;
double length;
final boolean active;

// INSERT CODE HERE that will not cause complie error

} Select the one correct answer.

a. instance { active = true; }

b. MyClass { gap += 5; }

c. { gap = 5; length = (active ? 100 : 200) + gap; }

d. { ; } **although valid but active needs to be initialised**

e. { length = 4.2; }

f. { active = (gap > 5); length = 5.5 + gap;}<<<<<<<<<<<<<answer

22.***

- **public class Initialization**
- **{**
- **private static String msg(String msg) {**
- **System.out.println(msg); return msg;**
- **}**
- **public Initialization() { m = msg("1"); }**
- **{**

```
        m = msg("2");          <<<<<<<<<<<<<<<<<<<if this would not have been there in instance initializer block this would have resulted in compile time error.
    }

    String m = msg("3");

    public static void main(String[] args) {
        Object obj = new Initialization();
    }

}
```

String Literal ---

A string literal is a reference to a String object. Since a string literal is a reference, it can be manipulated like any other String reference. i.e. it can be used to invoke methods of String class.

For example,

Int myLenght = "Hello world".lenght();

The Java language provides special support for the string concatenation operator (+), which has been overloaded for Strings objects. String concatenation is implemented through the StringBuffer class and its append method.

For example,

String finalString = "Hello" + "World"

Would be executed as

String finalString = new StringBuffer().append("Hello").append("World").toString();

The Java compiler optimizes handling of string literals. Only one String object is shared by all string having same character sequence. Such strings are said to be interned, meaning that they share a unique String object. The String class maintains a private pool where such strings are interned.

// splitting a string literal over more than one line

```
String s = "abcdefghijklmnopqrst"
        + "uvwxyz";
```

There is no speed penalty for the + concatenation. It is done at compile time. String literals can be used anywhere you might use a String reference. e.g. "abc".charAt(1) is legal. For problematic/awkward/ reserved/quotable characters like embedded ", see escape sequences below.

The Java compiler optimizes handling of string literals. Only one String object is shared by all string having same character sequence. Such strings are said to be interned, meaning that they share a unique String object. The String class maintains a private pool where such strings are interned.

For example,

```
String str1="Hello";
String str2="Hello";

If(str1 == str2) System.out.println("Equal");
```

Would print true when run.

Since the String objects are immutable. Any operation performed on one String reference will never have any effect on other references denoting the same object.

public String.intern() (JSK 1.3)

"Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.
When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned."

To save memory (and speed up testing for equality), Java supports "interning" of Strings. When the intern() method is invoked on a String, a lookup is performed on a table of interned Strings. If a String object with the same content is already in the table, a reference to the String in the table is returned. Otherwise, the String is added to the table and a reference to it is returned. The result is that after interning, all Strings with the same content will point to the same object. This saves space, and also allows the Strings to be compared using the == operator, which is much faster than comparison with the equals(Object) method.

Java automatically interns String literals. This means that in many cases, the == operator appears to work for Strings in the same way that it does for ints or other primitive values.

What is String literal pool?
The creation of two strings with the same sequence of letters without the use of the new keyword will create pointers to the same String in the Java String literal pool. The String literal pool is a way Java conserves resources.

To cut down the number of String objects created in the JVM, the String class keeps a pool of strings. Each time your code create a string literal, the JVM checks the string literal pool first. If the string already exists in the pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object instantiates, then is placed in the pool. Java can make this optimization since strings are immutable and can be shared without fear of data corruption.

String objects created with the new operator do not refer to objects in the string pool but can be made to using String's intern() method. The java.lang.String.intern() returns an interned String, that is, one that has an entry in the global String pool. If the String is not already in the global String pool, then it will be added.

```
public class Program
{
   public static void main(String[] args)
   {
      // Create three strings in three different ways.
      String s1 = "Hello";
      String s2 = new StringBuffer("He").append("llo").toString();
      String s3 = s2.intern();

      // Determine which strings are equivalent using the ==
      // operator
      System.out.println("s1 == s2? " + (s1 == s2));
      System.out.println("s1 == s3? " + (s1 == s3));
   }
}
```

The output is

```
s1 == s2? false
s1 == s3? true
```

There is a table always maintaining a single reference to each unique String object in the global string literal pool ever created by an instance of the runtime in order to optimize space. That means that they always have a reference to String objects in string literal pool, therefore, the string objects in the string literal pool not eligible for garbage collection.

String literals-or, more generally, strings that are the values of constant expressions-are "interned" so as to share unique instances, using the method String.intern.

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

```
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((other.Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");//<<<<
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
class Other { static String hello = "Hello"; }
```

produces the output:

true true true true false true

This example illustrates six points:

* Literal strings within the same class in the same package represent references to the same String object.
* Literal strings within different classes in the same package represent references to the same String object.
* Literal strings within different classes in different packages likewise represent references to the same String object.
* Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
* Strings computed by concatenation at run time are newly created and therefore distinct.

The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

http://www.xyzws.com/faq.do?cat=faq&article=3
http://javatechniques.com/public/java/docs/basics/string-equality.html

```
public boolean equals(Object o)
 {
  TreeNode tn=(TreeNode) o;
  if(o==null)
   return false;
  if(this.equals(o))  << Recursive call
   return true;

  return(tn.getValue().equals(this.getValue())) ;
 }
```

new

- two final classes: String and StringBuffer.
- only one String object is shared by all string-valued constant expressions with the same character sequence. Such strings are said to be interned, meaning that they share a unique String object if they have the same content. The String class maintains a private pool where such strings are interned.
- The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:
- `String str2 = "You cannot change me!";`
- `String str3 = "You cannot" + " change me!"; // Compile-time constant expression`
-

In the following code, both the references can1 and can2 denote the same String object that contains the string "7Up":

```
String can1 = 7 + "Up";  // Value of compile-time constant expression: "7Up"
String can2 = "7Up";     // "7Up"
```

However, in the code below, the reference can4 will denote a new String object that will have the value "7Up" at runtime:

```
String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```
new

The expression 7 + words is not a constant expression and, therefore, results in a new String object .

String Constructors

`String(String s)`

This constructor creates a new String object, whose contents are the same as those of the String object passed as argument.

`String()`

This constructor creates a new String object, whose content is the empty string, "".

Note that using a constructor creates a brand new String object, that is, using a constructor does not intern the string. A reference to an interned string can be obtained by calling the intern() method in the String class

Constructing String objects can also be done from arrays of bytes, arrays of characters, or string buffers:

```
byte[] bytes = {97, 98, 98, 97};
char[] characters = {'a', 'b', 'b', 'a'};
StringBuffer strBuf = new StringBuffer("abba");
//...
String byteStr = new String(bytes);      // Using array of bytes: "abba"
```
new
```
String charStr = new String(character);  // Using array of chars: "abba"
```
new
```
String buffStr = new String(strBuf);     // Using string buffer: "abba"
```
new

- Using the new operator with a String constructor always creates a new String object.

```
String byteStr = new String(bytes).intern();      //    What  ?
```

Comparing Strings

Characters are compared based on their integer values.

```
boolean test = 'a' < 'b';    // true since 0x61 < 0x62


int compareTo(String str2)
int compareTo(Object obj)
```

The first compareTo() method compares the two strings and returns a value based on the outcome of the comparison:

-
- the value 0, if this string is equal to the string argument
-
- a value less than 0, if this string is lexicographically less than the string argument
-
- a value greater than 0, if this string is lexicographically greater than the string argument

Character Case in a String

```
String toUpperCase()
String toUpperCase(Locale locale)

String toLowerCase()
String toLowerCase(Locale locale)
```

Note that the original string is returned if none of the characters need their case changed, but a new String object is returned if any of the characters need their case changed. new

Concatenation of Strings

Concatenation of two strings results in a string that consists of the characters of the first string followed by the characters of the second string. In addition, the following method can be used to concatenate two strings:

```
String concat(String str)
```

The concat() method does not modify the String object on which it is invoked, as String objects are immutable. Instead the concat() method returns a reference to a brand new String object: new

A simple way to convert any primitive value to its string representation is by concatenating it with the empty string (""), using the string concatenation operator (+)

```
String strRepresentation = "" + 2003;  // "2003"
```

Some more examples of string concatenation follow:

```
String motto = new String("Program once");      // (1) new, so new String object
motto += ", execute everywhere.";               // (2) resolved at runtime, no
new String
motto = motto.concat(" Don't bet on it!");      // (3) concat returns new String
ref
```

Searching for Characters and Substrings

If the search is unsuccessful, the value –1 is returned.

`int indexOf(int ch)`
Finds the index of the first occurrence of the argument character in a string.

`int indexOf(int ch, int fromIndex)`
Finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument.
If the index argument is negative, the index is assumed to be 0.
If the index argument is greater than the length of the string, it is effectively considered to be equal to the length of the string—returning the value -1.

`int indexOf(String str)`
Finds the start index of the first occurrence of the substring argument in a string.

`int indexOf(String str, int fromIndex)`
Finds the start index of the first occurrence of the substring argument in a string, starting at the index specified in the second argument.

Extracting Substrings

```
String trim()     //Returns a new String only when invoking object is different
than........
```

This method can be used to create a string where white space (in fact all characters with values less than or equal to the space character '\u0020') from the front (leading) and the end (trailing) of a string has been removed.

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)

//Returns a new String only when invoking object is different than........
```

Converting Primitive Values and Objects to Strings

The String class overrides the toString() method in the Object class and returns the String object itself:

String toString()

The String class also defines a set of static overloaded valueOf() methods to convert objects and primitive values into strings.

```
static String valueOf(Object obj)
static String valueOf(char[] character)
static String valueOf(boolean b)
static String valueOf(char c)

static String valueOf(int i)
static String valueOf(long l)
static String valueOf(float f)
static String valueOf(double d)
```

All these methods return a string representing the given parameter value.

```
String doubleStr = String.valueOf(Math.PI);                    //
"3.141592653589793"
```

- ('c' + 'o' + 'o' + 'l') // OK
- All wrapper classes are declared final.
-

Exam:

```
package fundamental;

public class Exam
{
        public String toString()
        {
                System.out.println("I m toString()");
                return null;
        }
        public static void main(String[] args)
        {
                String motto1 = "Program once";
                String motto2 = new String("Program once").intern();          // (1)   new, so
new String object
                String motto = new String("Program once");
                System.out.println("1: "+motto1==motto2);                          //
False
                System.out.println(motto1==motto2);                                    //
        True
                System.out.println(motto1==motto);                             //
False
                System.out.println("2: "+motto1.equals(motto2));                   //
2:true

                String motto3 = motto1.replace('o','o').intern();
                System.out.println("3: "+motto1==motto3);                          //
False
```

```
                System.out.println("4: "+"1"=="1");                                  //
False
                System.out.println("1"=="1");                                                //
        True
                System.out.println("4: "+"1"=="1".trim());                           //
False
                System.out.println("1"=="1".trim());                                 //
True

                String motto5 = "Program once";
                System.out.println(motto5==motto5.substring(0));                    //
True

                System.out.println("Hello".startsWith(""));                          //
True
                System.out.println("Hello".endsWith(""));                            //
True


        }

}
```

Get the o/p

1. System.out.println("1: "+motto1==motto2);
2.

The StringBuffer Class

In contrast to the String class, which implements immutable character strings, the StringBuffer class implements mutable character strings. Not only can the character string in a string buffer be changed, but the capacity of the string buffer can also change dynamically. The capacity of a string buffer is the maximum number of characters that a string buffer can accommodate before its size is automatically augmented.

- String and StringBuffer classes are two independent final classes, both directly extending the Object class.
- Hence, String references cannot be stored (or cast) to StringBuffer references and vice versa.
- Both String and StringBuffer are thread-safe.

Constructing String Buffers

The final class StringBuffer provides three constructors that create and initialize StringBuffer objects and set their initial capacity.

`StringBuffer(String s)`
The contents of the new StringBuffer object are the same as the contents of the String object passed as argument.
The initial capacity of the string buffer is set to the length of the argument string, plus room for 16 more characters.

`StringBuffer(int length)`
The new StringBuffer object has no content. The initial capacity of the string buffer is set to the value of the argument length, which cannot be less than 0.

`StringBuffer()`
This constructor also creates a new StringBuffer object with no content. The initial capacity of the string buffer is set for 16 characters.

Examples of StringBuffer object creation and initialization:

```
StringBuffer strBuf1 = new StringBuffer("Phew!");      // "Phew!", capacity 21
StringBuffer strBuf2 = new StringBuffer(10);           // "", capacity 10
StringBuffer strBuf3 = new StringBuffer();             // "", capacity 16
```

```
public class AStringQuestion
{
static String s1;
static String s2;
public static void main(String args[])
{
s2 = s1+s2;
System.out.println(s2);

System.out.println("args.length:  "+args.length);
}
}
```

---

nullnull
0

```
package fundamental;

import java.io.IOException;

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

		System.out.println(Math.min(Float.NaN, Float.POSITIVE_INFINITY));
		System.out.println(Math.max(Float.NaN, Float.POSITIVE_INFINITY));

		if (Math.max(Float.POSITIVE_INFINITY,Double.POSITIVE_INFINITY)==Double.
POSITIVE_INFINITY)
		{
			System.out.println("Float.POSITIVE_INFINITY");
		}
		else
		{
			System.out.println("Double.POSITIVE_INFINITY");
		}
	}
}
```

```
NaN
NaN
Float.POSITIVE_INFINITY



//	Float.POSITIVE_INFINITY == Double.POSITIVE_INFINITY  TRUE
```

- **All Wrrapper classe are final.**
- **The objects of the wrapper classes that can be instantiated are immutable.**
- **Although the Void class is considered a wrapper class, it does not wrap any primitive value and is not instantiable (i.e., has no public constructors). It just denotes the Class object representing the keyword void.**

CONSTRUCTORS

The Character class has only one public constructor, taking a char value as parameter.
The other wrapper *classes* all have two public one-argument constructors: one takes a primitive value and the other takes a string.

- **WrapperType(type v)**
- **WrapperType(String str)**

Converting Primitive Values to Wrapper Objects

A constructor that takes a primitive value can be used to create wrapper objects.

```
Character charObj1   = new Character('\n');
Boolean   boolObj1   = new Boolean(true);
Integer   intObj1    = new Integer(2003);
Double    doubleObj1 = new Double(3.14);
```

Converting Strings to Wrapper Objects

A constructor that takes a String object representing the primitive value, can also be used to create wrapper objects.
The constructors for the numeric wrapper types throw an unchecked NumberFormatException if the String parameter does not parse to a valid number.

```
Boolean boolObj2   = new Boolean("TrUe");       // case ignored: true
Boolean boolObj3   = new Boolean("XX");         // false wiil compile and run
successfully and will assign false to it


Integer intObj2    = new Integer("2003");
Float i=new Float("-InfiNity");Exception at run time case sensitive
Float i=new Float("-Infinity");valid


Double  doubleObj2 = new Double("3.14");
Long    longObj1   = new Long("3.14");          // NumberFormatException
```

Each wrapper class (except Character) defines the static method valueOf(String s) that returns the wrapper object corresponding to the primitive value represented by the String object passed as argument. This method for the numeric wrapper types also throws a

NumberFormatException if the String parameter is not a valid number.

Object CLASS

1. All classes extend the Object class, either directly or indirectly.
2. A class declaration, without the extends clause, implicitly extends the Object class**.**
3. **Methods**
   - **int hashCode()**::When storing objects in hash tables, this method can be used to get a hash value for an object. This value is guaranteed to be consistent during the execution of the program.
   - **hashCode method defined by class Object does return distinct integers for distinct objects**. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)
   - **boolean equals(Object obj):: ** returns true only if the two references compared denote the same object
   - **final Class getClass()**::Returns the runtime class of the object, which is represented by an object of the class java.lang.Class at runtime.
   - **String toString():::**If a subclass does not override this method, it returns a textual representation of the object, which has the following format: **"<name of the class>@<hash code value of object>"**
   - **protected void finalize() throws Throwable:::**
   - **protected Object clone() throws CloneNotSupportedException:::**
     i. **cloned objects** are exactly the same (i.e., have identical states) as the current object can be created by using the clone() method,
     ii. Primitive values and reference values are copied. **This is called shallow copying.**
     iii. **If the overriding clone() method relies on the clone() method in the Object class, then the subclass must implement the Cloneable marker interface** to indicate that its objects can be safely cloned. Otherwise, the clone() method in the Object class will throw a checked **CloneNotSupportedException.**

WrapperClass

1. Primitive values in Java are not objects.
2. TO manipulate these values as objects, the **java.lang** package provides a wrapper class for each of the primitive data types.
3. All wrapper classes are **final**.
4. The objects of all wrapper classes that can be instantiated are **immutable**.
5. **Void class is considered a wrapper class**, it does not wrap any primitive value and is **not instantiable** (i.e., has no public constructors).
6. The classes Byte, Short, Integer, Long, Float, and Double are numeric wrapper classes that extend the Number class.
7. **E**ach wrapper class **except char** has two constructors. taking a char value as parameter.

8. These are
9. WrapperType(**type v**)<<<<converting **primitive type to wrapper** object
10. WrapperType(**String str**)<<<converting **string to wrapper** object
11. The **constructors** for the **numeric wrapper types** throw **an unchecked NumberFormatException** if the String parameter does not parse to a valid number
12. **examples**
    a. Boolean boolObj2 = new Boolean("TrUe");// case ign: true
    b. Boolean boolObj3 = new Boolean("XX"); // **false**
    c. Integer intObj2 = new Integer("2003");
    d. Double doubleObj2 = new Double("3.14");
    e. Long longObj1 = new Long("3.9"); /**NumberFormatException**
    f. Byte byteObj2 = new Byte((byte) 16); // Cast mandatory
    g. Integer intObj5 = new Integer(42030);
    h. Double doubleObj4 = new Double(Math.PI);
13. **Common Wrapper Class Utility Methods**
14. **Static Methods**
    i. **static** WrapperType **valueOf**(**String s**)
        i. NOT Available in Character class.
        ii. **Returns wrapper object** corresponding to the primitive value represented by the String object(argument) .
        iii. **Example**
            1. Boolean boolObj4 = Boolean.valueOf("false");
            2. Integer intObj3 = Integer.valueOf("1949");
            3. Double doubleObj3 = Double.valueOf("-3.0");
        iv. Throws a **NumberFormatException** if the String parameter is not a valid number.
        v. **Integer wrapper types** ie Byte,Short,Integer,Long define an **overloaded** valueOf
15. **static** WrapperType valueOf(String s, **int base**)
        vi. **Example**
            1. Byte byteObj1 = Byte.valueOf("1010", 2); // Decimal value 10
            2. Short shortObj2 = Short.valueOf("012", 8); // Not "\012". Decimal value 10.
            3. Integer intObj4 = Integer.valueOf("-a", 16); // Not "-0xa". Decimal value -10.
            4. Long longObj2 = Long.valueOf("-a", 16); // Not "-0xa". Decimal value -10L

    j. static String toString(type v)
        i. returns the string corresponding to the **primitive value of type** passed as argument .
        ii. Example
            1. String charStr2 = Character.toString('\n'); // "\n"
            2. String boolStr2 = Boolean.toString(true); // "true"
            3. String intStr2 = Integer.toString(2003); // Base 10. "2003"
            4. String doubleStr2 = Double.toString(3.14); // "3.14"
        iii. For Floating point no.The NaN value, positive infinity and negative infinity will result in the strings "NaN", "Infinity", and "-Infinity"

, respectively

iv. The wrapper classes Integer and Long define **overloaded toString()** methods for converting integers to string representation in decimal, binary, octal, and hexadecimal notation .

v. **static** String **toString(int i, int base)**
- returns the minus sign '-' as the first character if the integer i is negative

vi. **static** String **toBinaryString**(int i)

vii. **static** String **toHexString**(int i)

viii. **static** String **toOctalString**(int i)

k. **type** parseType(**String** s)

i. **Available with numeric wrapper classes only**

ii. throw a NumberFormatException if the String parameter is not a valid argument

iii. Example

```
1. byte   value1 = Byte.parseByte("16");
2. int    value2 = Integer.parseInt("2010");      // parseInt, not parseInteger.
3. int    value3 = Integer.parseInt("7UP");       // NumberFormatException
4. double value4 = Double.parseDouble("3.14");
```

iv. **Overloaded version** for integertypes::byte,short,int,long

v. type parseType(String s, int base)

vi. Example

```
1. byte  value6 = Byte.parseByte("1010", 2);    // Decimal value 10
2. short value7 = Short.parseShort("012", 8);   // Not "\012". Decimal value 10.
3. int   value8 = Integer.parseInt("-a", 16);   // Not "-0xa". Decimal value -10.
4. long  value9 = Long.parseLong("-a", 16);     // Not "-0xa". Decimal value -10L.
```

13. Non-static utility methods

l. String toString()

i. Available in each wrapper class

ii. Overiden method of object class

m. type typeValue()

i. returns the primitive value in the wrapper object

ii. Example

```
1. char    c = charObj1.charValue();          // '\n'
2. boolean b = boolObj2.booleanValue();      // true
3. int     i = intObj1.intValue();           // 2003
4. double  d = doubleObj1.doubleValue();     // 3.14
```

n. type Value()

i. converting the primitive value in the wrapper object to a value of any numeric primitive type:

ii. Available with numeric wrapper classes only

iii. Byte, Short, Integer, Long, Float, and Double

1. byte byteValue()
2. short shortValue()
3. int intValue()
4. long longValue()
5. float floatValue()
6. double doubleValue()

iv. example

v. short shortVal = intObj5.shortValue(); // (1)
vi. long longVal = byteObj2.longValue();
vii. int intVal = doubleObj4.intValue(); // (2) Truncation
viii. double doubleVal = intObj5.doubleValue();

o. int compareTo(WrapperType obj2)

i. Not Available with Boolean

ii. Each wrapper class (except Boolean) also implements the Comparable interface

iii. Returns value >,<,=0 depending on weather primitive value in current obj and obj2

iv. Current object & argument obj2 must have the same WrapperType. Otherwise, a ClassCastException is thrown.

p. boolean equals(Object obj2)

i. compares two wrapper objects for object value equality

q. **int** hashCode()

14. Character.MIN_VALUE
15. Character.MAX_VALUE
16. Each numeric wrapper class defines an assortment of constants, including the minimum and maximum value of the corresponding primitive data type:

r. <**wrapper class name**>.MIN_VALUE
s. <**wrapper class name**>.MAX_VALUE

17. The Boolean class defines the following wrapper objects to represent the primitive values true and false, respectively:

t. Boolean.TRUE
u. Boolean.FALSE

18.

Math CLASS

1. final class
2. all methods are static
3. Misc. Rounding methods

a. abs

i. static int abs(int i)
ii. static long abs(long l)
iii. static float abs(float f)
iv. static double abs(double d)

b. max

i. static int max(int a, int b)
ii. static long max(long a, long b)
iii. static float max(float a, float b)

iv. static double max(double a, double b)

c. min

i. static int min(int a, int b)
ii. static long min(long a, long b)
iii. static float min(float a, float b)
iv. static double min(double a, double b)

- **Eaxmples**
  a. long ll = Math.abs(2010L); // 2010L
  b. double dd = Math.abs(-Math.PI); // 3.141592653589793
  c. double d1 = Math.min(Math.PI, Math.E); // 2.718281828459045
  d. long m1 = Math.max(1984L, 2010L); // 2010L
  e. int i1 = (int) Math.max(3.0, 4); // Cast required

d. static double ceil(double d)

i. The method **ceil()** returns the smallest double value that is greater than or equal to the argument d, and is equal to a mathematical integer.

e. static double floor(double d)

i. The method floor() returns the largest double value that is less than or equal to the argument d, and is equal to a mathematical integer.

f. static int round(float f)

g. static long round(double d)

h. Examples

Table 10.1. Applying Rounding Functions

Argument:	**7.0**	**7.1**	**7.2**	**7.3**	**7.4**	**7.5**	**7.6**	**7.7**	**7.8**	**7.9**	**8.0**
ceil:	7.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0
floor:	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
round:	7	7	7	7	7	8	8	8	8	8	8
Argument:	-7.0	-7.1	-7.2	-7.3	-7.4	-7.5	-7.6	-7.7	-7.8	-7.9	-8.0
ceil:	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-7.0	-8.0
floor:	-7.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0	-8.0
round:	-7	-7	-7	-7	-7	-7	-8	-8	-8	-8	-8

4. Exponential functions

i. static double pow(double d1, double d2)

i. The method pow() returns the value of d1 raised to the power of d2 (i. e., $d1^{d2}$).

j. static double exp(double d)

i. The method exp() returns the exponential number e raised to the power of d (i.e., e^{d}).

k. static double log(double d)

i. The method log() returns the natural logarithm (base e) of d (i.e., $\log_e d$).

l. static double sqrt(double d)

i. The method sqrt() returns the square root of d (i.e., $d^{0.5}$). **For a NaN or a negative argument, the result is a NaN**

m. Examples

i. double r = Math.pow(2.0, 4.0); // 16.0
ii. double v = Math.exp(2.0); // 7.38905609893065
iii. double l = Math.log(Math.E); // 0.9999999999999981
iv. double c = Math.sqrt(3.0*3.0 + 4.0*4.0); // 5.0

5. Trigonometry Functions

n. static double sin(double d)

i. The method sin() returns the trigonometric sine of an angle d specified in radians.

o. static double cos(double d)

i. The method cos() returns the trigonometric cosine of an angle d specified in radians.

p. static double tan(double d)

i. The method tan() returns the trigonometric tangent of an angle d specified in radians.

q. static double toRadians(double degrees)

i. The method toRadians() converts an angle in degrees to its approximation in radians.

r. static double toDegrees(double radians)

i. The method toRadians() converts an angle in radians to its approximation in degrees.

s. Examples

6. Pseudorandom Number Generator

t. static double random()

i. returns a random number **greater than or equal to 0.0** and **less than 1.0,** where the value is selected randomly from the range according to a uniform distribution.
ii. example
1. for (int i = 0; i < 10; i++)
2. System.out.println((int)(Math.random()*10)); // int **values in range [0 .. 9].**
iii. The loop will generate a run of ten pseudorandom integers between 0 (inclusive) and 10 (exclusive).

STRING CLASS

1. Final class
2. string literal: String str1 = "You cannot change me!";
3. string literal is a reference to a String object
4. The compiler optimizes handling of string literals (and compile-time constant expressions that evaluate to strings):
5. only one String object is shared by all string-valued constant expressions with the same character sequence.
6. Such strings are said to be interned, meaning that they share a unique String object if they have the same content.
7. The String class maintains a private pool where such strings are interned.
8. The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:
9. The compile-time evaluation of the constant expression involving the two string literals, results in a string that is already interned:

a. Example

i. String str3 = "You cannot" + " change me!"; // Compile-time constant expression
ii. String can1 = 7 + "Up"; // Value of compile-time constant expression: "7Up"
iii. String word = "Up";
iv. String can4 = 7 + word; // Not a compile-time constant expression.

10. using a constructor creates a brand new String object, that is, using a constructor does not intern the string.
11. A reference to an interned string can be obtained by calling the intern() method in the String class—in practice, there is usually no reason to do so
12. Constructing String objects can also be done from **arrays of bytes, arrays of characters, or string buffers**
    b. byte[] bytes = {97, 98, 98, 97};
    c. char[] characters = {'a', 'b', 'b', 'a'};
    d. StringBuffer strBuf = new StringBuffer("abba");
    e. String byteStr = new String(bytes); // Using array of bytes: "abba"
    f. String charStr = new String(character); // Using array of chars: "abba"
    g. String buffStr = new String(strBuf); // Using string buffer: "abba"

13. The Java language provides special support for the string concatenation operator (+), which has been overloaded for Strings objects.
14. String concatenation is implemented through the StringBuffer class and its append method
    h. example,
        i. String finalString = "Hello" + "World" Would be executed as
        ii. String finalString = new StringBuffer().append("Hello").append("World").toString();
15. **Reading Characters from a String**
    i. char charAt(int index)
        i. first character is at index 0 and the last one at index one less than the number of characters in the string.
        ii. If the index value is not valid, a **StringIndexOutOfBoundsException** is thrown.
    j. void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
        i. copies characters from the current string into the destination character array. Characters from the current string are read from index srcBegin to the index srcEnd-1, inclusive. They are copied into the destination array, starting at index dstBegin and ending at index dstbegin+(srcEnd-srcBegin)-1. The number of characters copied is (srcEnd-srcBegin).
        ii. An **IndexOutOfBoundsException** is thrown if the indices do not meet the criteria for the operation.
    k.
16. **Comparing Strings>>>compares String lexicographically**
    l. int compareTo(String str2)
    m. int compareTo(Object obj)>>throws classcast exception if obj is not String
17. **Searching for Characters and Substrings**
    n. int indexOf(int ch)
        i. Finds the index of the first occurrence of the argument character in a string.
    o. int indexOf(int ch, int fromIndex)
        i. Finds the index of the first occurrence of the argument character in a string, starting at the index specified in the second argument.
        ii. If **the index argument is negative, the index is assumed**

to be 0

iii. If the index argument is greater than the length of the string, it is effectively considered to be equal to the length of the string—returning the value -1.

p. int indexOf(String str)

i. Finds the start index of the first occurrence of the substring argument in a string.

q. int indexOf(String str, int fromIndex)

i. Finds the start index of the first occurrence of the substring argument in a string, starting at the index specified in the second argument.

r. int lastIndexOf(int ch)

s. int lastIndexOf(int ch, int fromIndex)

t. int lastIndexOf(String str)

u. int lastIndexOf(String str, int fromIndex)

i. search is backward toward the start of the string.

ii. the index of the last occurrence of the character or substring is found.

v. String replace(char oldChar, char newChar)

18. **Extracting Substrings**

w. String substring(int startIndex)

x. dString substring(int startIndex, int endIndex)

i. the last character in the substring is at index endIndex-1.

19. **Converting Primitive Values and Objects to Strings**

y. toString()

z. static String valueOf(Object obj)

aa. static String valueOf(char[] character)

bb. static String valueOf(boolean b)

cc. static String valueOf(char c)

i. Equivalent to toString toString()

STRING BUFFER

1. Mutable
2. does not override equals & hashcode
3. Both String and StringBuffer are thread-safe.
4. Not only can the character string in a string buffer be changed,
5. capacity of the string buffer can also change dynamically.
6. The capacity of a string buffer is the maximum number of characters that a string buffer can accommodate before its size is automatically augmented.
7. **Constructing String Buffers/Constructors**

a. StringBuffer(String s)

i. The contents of the new StringBuffer object are the same as the contents of the String object passed as argument.

ii. The initial capacity of the string buffer is set to the length of the argument string, plus room for 16 more characters.

b. StringBuffer(int length)

i. The new StringBuffer object has no content.

ii. The initial capacity of the string buffer is set to the value of the **argument length, which cannot be less than 0.**

c. StringBuffer()

i. This constructor also creates a new StringBuffer object with no content.

ii. The initial capacity of the string buffer is set for 16 characters.

1. StringBuffer strBuf1 = new StringBuffer("Phew!"); // "Phew!", capacity 21
2. StringBuffer strBuf2 = new StringBuffer(10); // "", capacity 10
3. StringBuffer strBuf3 = new StringBuffer(); // "", capacity 16

8. **Reading and Changing Characters in String Buffers**
   - **d. int length()**
     i. Returns the number of characters in the string buffer.
   - **e. char charAt(int index)**
   - **f. void setCharAt(int index, char ch)**
     i. A **StringIndexOutOfBoundsException** is thrown **if the index is not valid.**
9. **Appending Characters to a String Buffer**
   - **g. StringBuffer append(Object obj)**
   - **h. StringBuffer append(String str)**
   - **i. StringBuffer append(char[] str)**
   - **j. StringBuffer append(char[] str, int offset, int len)**
   - **k. StringBuffer append(char c)**
   - **l. Buffer append(boolean b)**
   - **m. StringBuffer append(int i)**
   - **n. StringBuffer append(long l)**
   - **o. StringBuffer append(float f)**
   - **p. StringBuffer append(double d)**
     i. These methods convert the primitive value of the argument to a string by applying the static method String.valueOf() to the argument, before appending the result to the string buffer:
10. **Inserting Characters in a String Buffer**
    q. The **overloaded** method **insert**() can be used to insert characters at a given position in a string buffer.zx
       i. StringBuffer insert(int offset, Object obj)
       ii. StringBuffer insert(int offset, String str)
       iii. StringBuffer insert(int offset, char[] str)
       iv. StringBuffer insert(int offset, char c)
       v. StringBuffer insert(int offset, boolean b)
       vi. StringBuffer insert(int offset, int i)
       vii. StringBuffer insert(int offset, long l)
       viii. StringBuffer insert(int offset, float f)
       ix. StringBuffer insert(int offset, double d)
       1. The argument is converted, if necessary, by applying the static method String.valueOf().
       2. The offset argument specifies where the characters are to be inserted and must be greater than or equal to 0.
11. **Deleting Characters in a String Buffer**
    - **r. StringBuffer deleteCharAt(int index)**
      i. The first method deletes a character at a specified index in the string buffer, contracting the string buffer by one character.
    - **s. StringBuffer delete(int start, int end)**
      i. The second method deletes a substring, which is specified by the **start index (inclusive) and the end index (exclusive).**
12. **Controlling String Buffer Capacity**

t. int capacity()

i. Returns the current capacity of the string buffer, that is, the number of characters the current buffer can accommodate without allocating a new, larger array to hold characters.

u. void ensureCapacity(int minCapacity)

i. Ensures that there is room for at least minCapacity number of characters. It expands the string buffer, depending on the current capacity of the buffer.

v. void setLength(int newLength)

i. This method ensures that the actual number of characters, that is, length of the string buffer, is exactly equal to the value of the newLength argument, which must be greater than or equal to 0. **This operation can result in the string being truncated or padded with null characters ('\u0000').**

ii. This method only affects the capacity of the string buffer if the value of the parameter newLength is greater than current capacity.

iii. One use of this method is to clear the string buffer

iv. buffer.setLength(0); // Empty the buffer.

```
public class MyClass {
   public static void main(String[] args) {
      String s = "hello";
      StringBuffer sb = new StringBuffer(s);
      sb.reverse();
     if (s == sb) System.out.println("a");//complie time error
                  It compares references of two classes that are not related.
      if (s.equals(sb)) System.out.println("b");??will it throw classcast exception at
run time
      if (sb.equals(s)) System.out.println("c");
   }
}
public class StringMethods {
   public static void main(String[] args) {
      String str = new String("eenny");
      str.concat(" meeny");
      StringBuffer strBuf = new StringBuffer(" miny");
      strBuf.append(" mo");
      System.out.println(str + strBuf);
   }
}
```

13. The code will fail to compile.
14. The program will print eenny meeny miny mo when run.
15. The program will print meeny miny mo when run.
16. The program will print eenny miny mo when run.
17. The program will print eenny meeny miny when run.

```
public Class Program{

    public static void main(String[] args){

       int   i = 0;

       int[] a = {1,2};

       a[i] = i = 2;

       System.out.println(i + " " + a[0]+ " "+a[1]);
    }

}
```

What is the output from the above code?

The output is:

2 2 2

Why does "int[] a={1,2}; a[i]=i=2;" run without throwing an ArrayIndexOutOfBoundsException?

You may raise some questions: "When does i get a value 2?", "Why doesn't it use the same value (i=2) for a[i] (which should actually give an ArrayOutOfBoundsException Exception)?", and so on.

The reason no ArrayIndexOutOfBoundsException is thrown is because each of the operands is evaluated from left to right. Basically the array reference is evaluated first then the rest of the expression is done according to the rules in the "15.26.1 Simple Assignment Operator =" of Java Language Specification 3rd Edition. Therefore, we first
a[i] while i is still 0. The assignment takes place after the leftmost operand is evaluated. Thereafter the assignment operator takes over and changes the value of i as well as a[0] to 2.

The assignment is performed from right to left, but operand evaluation is not the same as assignment.

We have the following expression:

a[i] = i = 2;

The first thing that happens is evaluating any operands that need evaluating. In this case, only one operand really needs evaluating - the first one. So, after evaluating that, the expression became:

a[0] = i = 2;

Then, assignment happens now, from right to left. The 2 is assigned to i, then assign to a[0]. So the expression:

a[0] = i = 2;

is equal to the following two expressions:

i = 2;
a[0] = i;

Hence, a[0] obtains a value of 2 and no an ArrayIndexOutOfBoundsException is thrown.

http://java.sun.com/docs/books/jls/second_edition/html/lexical.doc.html

Declaring Transient Variables

transient variables are not part of the object's persistent state. ==> A transient variable is a variable that cannot be serialized.
Transient variable can't be serialize. For example if a variable is declared as transient in a Serializable class and the class is written to an ObjectStream, the value of the variable can't be written to the stream instead when the class is retrieved from the ObjectStream the value of the variable becomes null.

```
{

        ...
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);

        MySerializableCls msc= new MySerializableCls(123, 2);
        oos.writeObject(msc);

        ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bais);

        MySerializableCls msc1 = (MySerializableCls) ois.readObject();

        System.out.println("msc:"+msc);
        System.out.println("msc1:"+msc1);
    }
}

class MySerializableCls implements Serializable{
        MySerializableCls(){}
        MySerializableCls(int _i, int _count){
            this.i = _i;
            this.count = _count;
        }

        public String toString(){
            return ("i: "+this.i + ";  count:"+this.count);
        }

        int i;
        transient int count;
}
```

Declaring Volatile Variables

Volatile Variable Declaration
volatile means that the variabe is modified asynchronously

If your class contains a member variable that is modified asynchronously by concurrently running threads, you can use Java's volatile keyword to notify the Java runtime system of this.

Java runtime system will use this information to ensure that the volatile variable is loaded from memory before each use, and stored to memory after each use thereby ensuring that the value of the variable is consistent and coherent within each thread.

volatile fields can be slower than non-volatile fields, because the system is forced to store to memory rather than use registers. But they may useful to avoid concurrency problems.
Volatile variables are always synchronized with the main memory copy.

The following variable declaration is an example of how to declare that a variable can be modified asynchronously by concurrent threads:

```
class VolatileExample {
   volatile int counter;
   . . .
}
```

The volatile modifier requests the Java VM to always access the shared copy of the variable so the its most current value is always read. If two or more threads access a member variable, AND one or more threads might change that variable's value, AND ALL of the threads do not use synchronization (methods or blocks) to read and/or write the value, then that member variable must be declared volatile to ensure all threads see the changed value.

http://www.javaperformancetuning.com/tips/volatile.shtml
http://journals.ecs.soton.ac.uk/java/tutorial/java/javaOO/variables.html

Class /Method Declaration

1. A class declaration introduces a new reference type.

   a. Syntax>>.class

   i. **<class modifiers>** `class` <class name>
   <extends clause> <implements clause> `// Class header`
   `{ // Class body`
   <field declarations>
   <method declarations>
   <nested class declarations>
   <nested interface declarations>
   <constructor declarations>
   <initializer blocks>
   `}`

   class header can specify the following information:

   ii. scope or accessibility modifier

   iii. additional class modifiers

   iv. any class it extends

   v. any interfaces it implements

   b. Syntax>> method

   i. <**method modifiers**> <**return type**> <method name> `(`< **formal parameter list**>`)`
   <throws clause> `// Method prototype`
   `{ // Method body`
   <local variable declarations>
   <nested local class declarations>
   <statements>
   `}`

   ii. In addition to the name of the method, the method prototype can specify the following information:

   iii. scope or accessibility modifier

   iv. additional method modifiers

   v. type of the return value, or `void` if the method does not return any value

   vi. formal parameter list

   vii. checked exceptions thrown by the method in a `throws` clause

   viii. The signature of a method comprises the method name and the formal parameter list only.

2. Class names and method names exist in different namespaces. Thus, there are no name conflicts

3. Modifiers for classes

    c. Public

    d. Default/friendly

    e. Abstract

    f. Final

4. Abstract class cannot be both final and abstract

5. Interfaces, which are inherently `abstract`, thus cannot be declared `final`

6. Statements can be grouped into various categories.

    g. Variable declarations with explicit initialization of the variables are called **declaration statements .**

    h. Control flow statements

    i. expression statements.

7. An expression statement is an expression terminated by a semicolon.

8. The expression is evaluated for its side effect and its value discarded.

9. Only certain types of expressions have meaning as statements.

10. They include the following:

    j. Assignments

    k. Increment and decrement operators

    l. Method calls

    m. Object creation expression with the `new` operator

11. A solitary semicolon denotes the empty statement that does nothing.

12. A block, `{}`, is a **compound statement** which can be used to group zero or more local declarations and statements (see Section 4.5, p. 123).

13. Blocks can be nested, since a block is a statement that can contain other statements.

14. A block can be used in any context where a simple statement is permitted.

15. **`Modifiers for methods`**

`public`	Accessible everywhere.
`protected`	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.

default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
`private`	Only accessible in its own class and not anywhere else.

16.

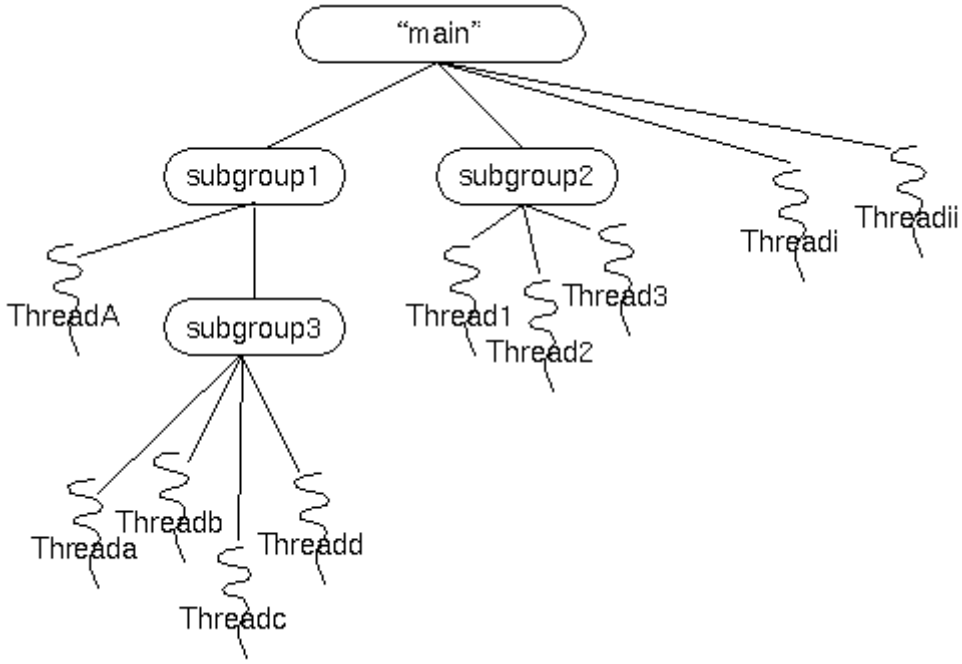
Table 4.5. Summary of Other Modifiers for Members

Modifiers	Fields	Methods
`static`	Defines a class variable.	Defines a class method.
`final`	Defines a constant.	The method cannot be overridden.
`abstract`	N/A	No method body is defined. Its class must also be designated `abstract`.
`synchronized`	N/A	Only one thread at a time can execute the method.
`native`	N/A	Declares that the method is implemented in another language.
`transient`	The value in the field will not be included when the object is serialized.	Not applicable.
`volatile`	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

The ThreadGroup Class

The ThreadGroup class manages groups of threads for Java applications. A ThreadGroup can contain any number of threads. The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.

ThreadGroups can contain not only threads but also other ThreadGroups. The top most thread group in a Java application is the thread group named "main". You can create threads and thread groups in the "main" group. You can also create threads and thread groups in subgroups of "main" and so on. The result is a root-like hierarchy of threads and thread groups.



http://tns-www.lcs.mit.edu/manuals/java-tutorial/java/threads/threadgroup.html#collection

Thread

```
public Thread(ThreadGroup group,
              Runnable target,
              String name,
              long stackSize)
```

Thread
>>> SecurityException
checkAccess<<< Search ... MultiThreaded Programming with JAVA.pdf

Thread State--
not-Runnable States
1. Sleep
2. Wait
    1. completion of IO
    2. completion of someOther thread (JOIN)
    3. signal (Notification)
    4. lock

Child Thread Inherits the parent's Priority

thread.setPriority(NORM_PRIORITY) >>> internally any access to the thread will call checkAcess().. that will call the checkAcess of the SecurityManager, if installed. Throws SecurityException

Priority >> min of threadGroup's prority and the specified priority

SecurityManager<<<<

Yield() >>> does the thread releases the acquired locks?
Yield()>>> preemptes the CPU... for anyother thread waiting for the CPU>>> Doesn't Gurantee that an expected thread will be executed.
If a thread is holding a lock (because it's in a synchronized method or block of code), it does not release the lock when it calls yield().

sleep(...) >> locks are held, while in sleep
sleep(mili, nano)
sleep(milli)
>>> InterruptedException

Object functions invoked by thread:
on some shared object. <<<
wait()>> wait(), wait(timeOut), wait(timeOutMilli, timeOutNano);
notify()
comes out on call of a notify(), timeOut, interrupt()
>>> wait() made the thread release the lock on that sharedObject.<<non of the other>>

invoked only on the objects, whose lock the thread holds. >> otherwise, it will result in throw of an IllegalMonitorStateException
wait() >>> wait for Notification()>> blocked for Notification()

a call to notify() is just a signal to move one of the thread blocked for the notification on this object into a "Block of lock acquisition State". This call is just a signal. The calling thread doesn't release the lock on the object. It will release it according to its convinience.

Conditions that wake a waiting thread:
1. notify() call on the object()
2. wait timeOut
3. interrupt()

>> interrupt()<<<<<

Join << Wait for a thread to die:
>>InterruptedException
thread1.join(); << its not a static
....<<< won't execute this untill the theard1 dies

join()

join(timeIOut)
join(timeOutMilli, timeOutNano)

http://www.witscale.com/scjp_studynotes/chapter8.html
http://gee.cs.oswego.edu/dl/cpj/mechanics.html
http://www-128.ibm.com/developerworks/library/j-thread.html

9.1 Multitasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking

At the coarse-grain level there is process-based multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running the spreadsheet program while also working with the word-processor. At the fine-grain level there is thread-based multitasking, which allows parts of the same program to run concurrently on the computer. A familiar example is a word-processor that is printing and formatting text at the same time. The sequence of code executed for each task defines a separate path of execution, and is called a thread (of execution).

In a single-threaded environment only one task at a time can be performed. CPU cycles are wasted, for example, when waiting for user input. Multitasking allows idle CPU time to be put to good use.

Some advantages of thread-based multitasking as compared to process-based multitasking are

- threads share the same address space
- context switching between threads is usually less expensive than between processes
- cost of communication between threads is relatively low

9.1 Overview of Threads

- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code, that is, they are lightweight compared to processes.

Every thread in Java is created and controlled by a unique object of the java.lang.Thread class.

The Main Thread

The runtime environment distinguishes between user threads and daemon threads. As long as a user thread is alive, the JVM does not terminate. A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program. Daemon threads exist only to serve user threads.

When a standalone application is run, a user thread is automatically created to execute the main() method. This thread is called the main thread. If no other user threads are spawned, the program terminates when the main() method finishes executing. All other threads, called child threads, are spawned from the main thread, inheriting its user-thread status. The main() method can then finish, but the program will keep running until all the user threads have finished. Calling the

setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started. Any attempt to change the status after the thread has been started, throws an IllegalThreadStateException. Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.

9.3 Thread Creation

A thread in Java is represented by an object of the Thread class. Implementing threads is achieved in one of two ways:

- implementing the java.lang.Runnable interface
- extending the java.lang.Thread class

Implementing the Runnable Interface

The Runnable interface has the following specification, comprising one method prototype declaration:

```
public interface Runnable {
    void run();
}
```

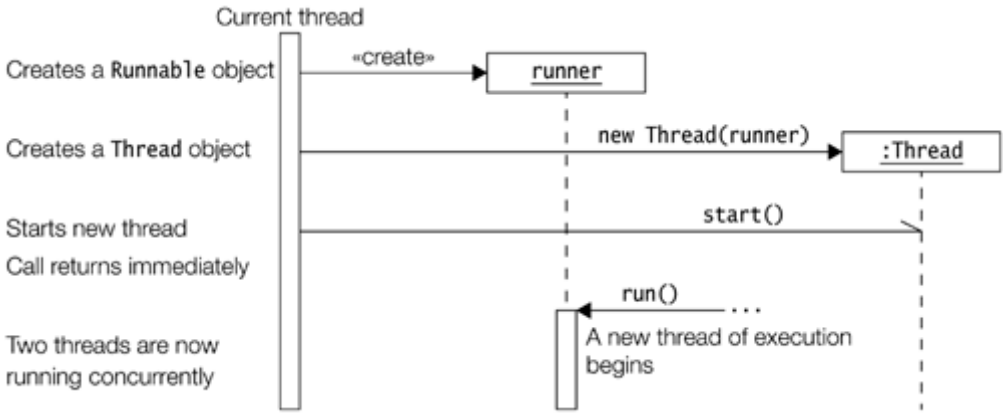
A thread, which is created based on an object that implements the Runnable interface, will execute the code defined in the public method run(). In other words, the code in the run() method defines an independent path of execution and thereby the entry and the exits for the thread. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.
The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.

2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run () method.

3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.

The run() method of the Runnable object is eventually executed by the thread represented by the Thread object on which the start() method was invoked.



The following is a summary of important constructors and methods from the java.lang.

Thread class:

`Thread(Runnable threadTarget)`
`Thread(Runnable threadTarget, String threadName)`
The argument threadTarget is the object whose run() method will be executed when the thread is started. The argument threadName can be specified to give an explicit name for the thread, rather than an automatically generated one. A thread's name can be retrieved by using the getName() method.

`static Thread currentThread()`
This method returns a reference to the Thread object of the currently executing thread.

`final String getName()`
`final void setName(String name)`
The first method returns the name of the thread. The second one sets the thread's name to the argument.

`void run()`
The Thread class implements the Runnable interface by providing an implementation of the run() method. This implementation does nothing. Subclasses of the Thread class should override this method.
If the current thread is created using a separate Runnable object, then the Runnable object's run() method is called.

`final void setDaemon(boolean flag)`
`final boolean isDaemon()`
The first method sets the status of the thread either as a daemon thread or as a user thread, depending on whether the argument is true or false, respectively. The status should be set before the thread is started. The second method returns true if the thread is a daemon thread, otherwise, false.

`void start()`
This method spawns a new thread, that is, the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an IllegalThreadStateException if the thread was already started.

Example 9.1 Implementing the Runnable Interface

```
class Counter implements Runnable {

    private int currentValue;

    private Thread worker;

    public Counter(String threadName) {
        currentValue = 0;
        worker = new Thread(this, threadName);   // (1) Create a new thread.
        System.out.println(worker);
        worker.start();                          // (2) Start the thread.
    }

    public int getValue() { return currentValue; }

    public void run() {                          // (3) Thread entry point
        try {
            while (currentValue < 5) {
                System.out.println(worker.getName() + ": " + (currentValue++));
                Thread.sleep(250);               // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
```

```
            System.out.println(worker.getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + worker.getName());
    }
}

public class Client {
    public static void main(String[] args) {
        Counter counterA = new Counter("Counter A"); // (5) Create a thread.

        try {
            int val;
            do {
                val = counterA.getValue();       // (6) Access the counter
value.
                System.out.println("Counter value read by main thread: " +
val);
                Thread.sleep(1000);              // (7) Current thread sleeps.
            } while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
        }

        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

The first line of the output shows the string representation of a Thread object: its name (Counter A), its priority (5), and its parent thread (main).

Extending the Thread Class

A class can also extend the Thread class to create a thread. A typical procedure for doing this is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.

2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.

3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

The Thread class implements the Runnable interface, which means that this approach is not much

different from implementing the Runnable interface directly. The only difference is that the roles of the Runnable object and the Thread object are combined in a single object.

The static method currentThread() in the Thread class can be used to obtain a reference to the Thread object associated with the current thread.

```
Thread.currentThread().getName());
```

Adding the following statement before the call to the start() method:

```
setDaemon(true);
```

Illustrates the daemon nature of threads. The program execution will now terminate after the main thread has completed, without waiting for the daemon Counter threads to finish normally.

Example 9.2 Extending the Thread Class

```
class Counter extends Thread {

    private int currentValue;

    public Counter(String threadName) {
        super(threadName);                      // (1) Initialize thread.
        currentValue = 0;
        System.out.println(this);
        start();                                // (2) Start this thread.
    }

    public int getValue() { return currentValue; }
    public void run() {                         // (3) Override from
superclass.
        try {
            while (currentValue < 5) {
                System.out.println(getName() + ": " + (currentValue++));
                Thread.sleep(250);              // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + getName());
    }
}

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

        System.out.println("Method main() runs in thread " +
                Thread.currentThread().getName());   // (5) Current thread

        Counter counterA = new Counter("Counter A"); // (6) Create a thread.
        Counter counterB = new Counter("Counter B"); // (7) Create a thread.

        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
```

```
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

- Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.

- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

Inner classes are useful for implementing threads that do simple tasks. The anonymous class below will create a thread and start it:

```
(   new Thread() {
        public void run() {
            for(;;) System.out.println("Stop the world!");
        }
    }
).start();
```

9.4 Synchronization

Threads share the same memory space, that is, they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently amongst several users without proper discipline, will jeopardize the integrity of the account data. Java provides high-level concepts for synchronization in order to control access to shared resources.

Locks

A lock (a.k.a. monitor) is used to synchronize access to a shared resource. A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource. At any given time, at the most one thread can hold the lock (i.e., own the monitor) and thereby have access to the shared resource. A lock thus implements mutual exclusion (a.k.a. mutex).

In Java, all objects have a lock—including arrays. This means that the lock from any Java object can be used to implement mutual exclusion. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the object lock.

The object lock mechanism enforces the following rules of synchronization:

- A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource. If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.

- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the java.lang.Class object associated with the class. Given a class A, the reference A.class denotes this unique Class object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The keyword synchronized and the lock form the basis for implementing synchronized execution of code. There are two ways in which execution of code can be synchronized:

- synchronized methods

- synchronized blocks

Synchronized Methods

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword synchronized. A thread wishing to execute a synchronized method must first obtain the object's lock (i.e., hold the lock) before it can enter the

object to execute the method. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits. No particular action on the part of the program is necessary. A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed.

Non-synchronized updating of the value in a field (Static / Non-Static) between the two threads is a disaster waiting to happen. This is an example of what is called a race condition. It occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state.

While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait. This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object. Such a method can invoke other synchronized methods of the object without being blocked. The non-synchronized methods of the object can of course be called at any time by any thread.

Static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method, proceeds analogous to that of an object lock for a synchronized instance method. A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any such methods in the same class. This, of course, does not apply to static, non-synchronized methods, which can be invoked at any time. A thread acquiring the lock of a class to execute a static synchronized method, has no bearing on any thread acquiring the lock on any object of the class to execute a synchronized instance method. In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.

Synchronized Blocks

Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object. The general form of the synchronized statement is as follows:

synchronized (<object reference expression>) { <code block> }

The **<object reference expression>** must evaluate to a **non-null reference value**, otherwise, a **NullPointerException** is thrown. The code block is usually related to the object on which the synchronization is being done. This is the case with synchronized methods, where the execution of the method is synchronized on the lock of the current object:

```
public Object pop()
{
    synchronized (this)
    {             // Synchronized block on current object
                  // ...
    }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown.

Object specification in the synchronized statement is mandatory. A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block. The braces of the block cannot be left out, even if the code block has just one statement.

```
class SmartClient
{
    BankAccount account;
    // ...
    public void updateTransaction()
    {
        synchronized (account)
        {
                  // (1) synchronized block
                  account.update();          // (2)
        }
    }
}
```

In the previous example, the code at (2) in the synchronized block at (1) is synchronized on the BankAccount object. If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time, only after synchronizing on the BankAccount object associated with this particular instance of SmartClient.

Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter. This is illustrated in the following code where the synchronized block at (5) uses the special form of the this reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method setPi() in an inner object can only access the private double field myPi at (2) in the synchronized block at (5), by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

```
class Outer
{                                        // (1) Top-level Class
    private double myPi;                 // (2)

    protected class Inner                // (3) Non-static member Class
    {

        public void setPi()              // (4)
        {

            synchronized(Outer.this)     // (5) Synchronized block on outer
object
            {

                  myPi = Math.PI;        // (6)
            }
        }
    }
}
```

Synchronized blocks can also be specified on a class lock:

synchronized (<class name>.class) { <code block> }

The block synchronizes on the lock of the object denoted by the reference <class name>.class. A static synchronized method classAction() in class A is equivalent to the following declaration:

```
static void classAction()
{
      synchronized (A.class)
      {
            // Synchronized block on class A
            // ...
      }
}
```

In summary, a thread can hold a lock on an object

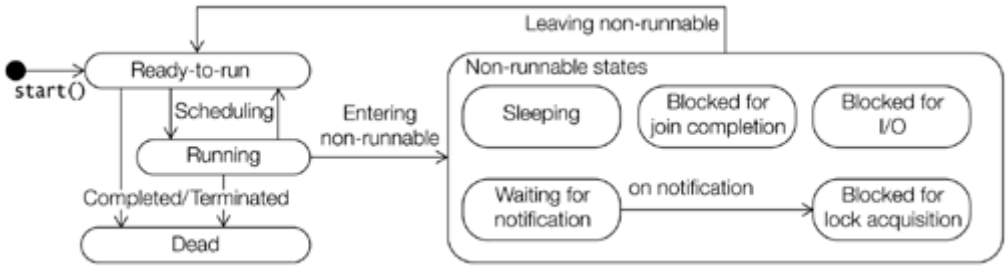
- by executing a synchronized instance method of the object
- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class

Thread States

Threads can exist in different states. Just because a thread's start() method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed.

- **Ready-to-run state:** A thread starts life in the Ready-to-run state.
- **Running state:** If a thread is in the Running state, it means that the thread is currently executing.
- **Dead state:** Once in this state, the thread cannot ever run again.
- **Non-runnable states:** A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.
  The non-runnable states can be characterized as follows:
  - **Sleeping:** The thread sleeps for a specified amount of time.
  - **Blocked for I/O:** The thread waits for a blocking operation to complete.
  - **Blocked for join completion:** The thread awaits completion of another thread.
  - **Waiting for notification:** The thread awaits notification from another thread.
  - **Blocked for lock acquisition:** The thread waits to acquire the lock of an object.

Figure: Thread States



Various methods from the Thread class are presented next.

```
final boolean isAlive()
```
This method can be used to find out if a thread is alive or dead. A thread is alive if it has been started but not yet terminated, that is, it is not in the Dead state.

```
final int getPriority()
```

`final void setPriority(int newPriority)`
The first method returns the priority of the current thread. The second method changes its priority. The priority set will be the minimum of the two values: the specified newPriority and the maximum priority permitted for this thread.

`static void yield()`
This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute.

`static void sleep (long millisec) throws InterruptedException`
The current thread sleeps for the specified time before it takes its turn at running again.

`final void join() throws InterruptedException`
`final void join(long millisec) throws InterruptedException`
A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively.

`void interrupt()`
The method interrupts the thread on which it is invoked. In the Waiting-for-notification, Sleeping, or Blocked-for-join-completion states, the thread will receive an InterruptedException.

Thread Priorities

Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state.

Priorities are integer values from 1 (lowest priority given by the constant Thread. MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5 (Thread.NORM_PRIORITY).

A thread inherits the priority of its parent thread. Priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class.

Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

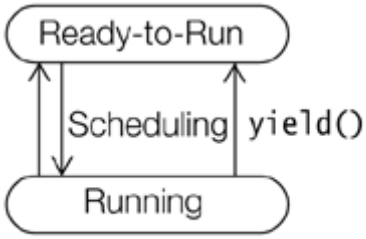
- Preemptive scheduling.
- If a thread with a higher priority than the current running thread moves to the Ready-to-run state, then the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.
- Time-Sliced or Round-Robin scheduling.
- A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

It should be pointed out that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Running and Yielding

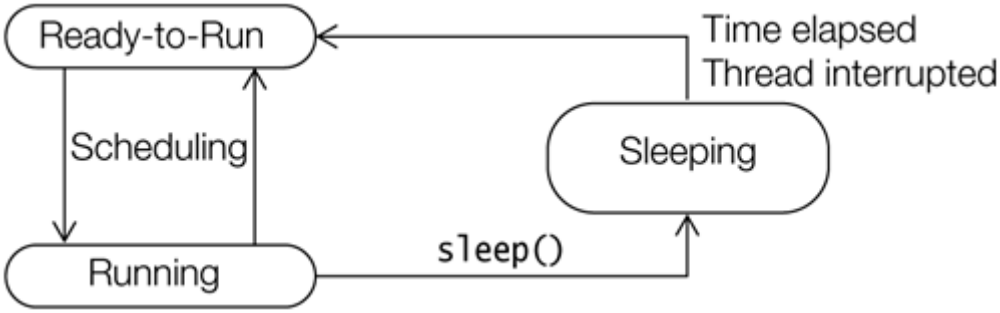
After its start() method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, that is, it waits for its turn to get CPU time. The thread scheduler decides which thread gets to run and for how long.

A call to the static method yield(), defined in the Thread class, will cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute.



By calling the static method yield(), the running thread gives other threads in the Ready-to-run state a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of canceling it by clicking on a Cancel button. If the computation thread hogs the CPU and the user clicks the Cancel button, chances are that it might take a while before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation.

Sleeping and Waking up



A call to the static method sleep() in the Thread class will cause the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before transitioning to the Ready-to-run state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an InterruptedException when it awakes and gets to execute.

There are serveral overloaded versions of the sleep() method in the Thread class.

Waiting and Notifying

Waiting and notifying provide means of communication between threads that synchronize on the same object. The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose. These final methods are defined in the Object class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException.

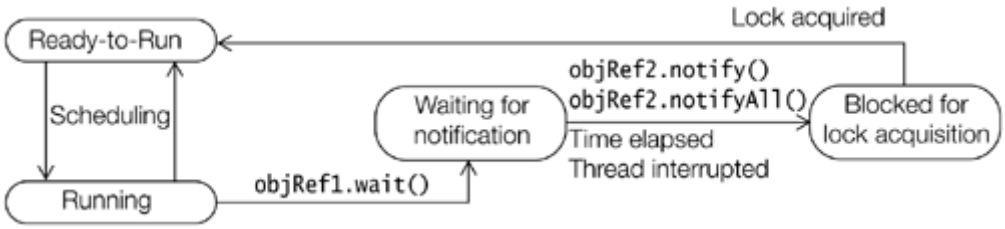
```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

A thread invokes the wait() method on the object whose lock it holds. The thread is added to the wait set of the object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

Communication between threads is facilitated by waiting and notifying. A thread usually calls the wait() method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock.



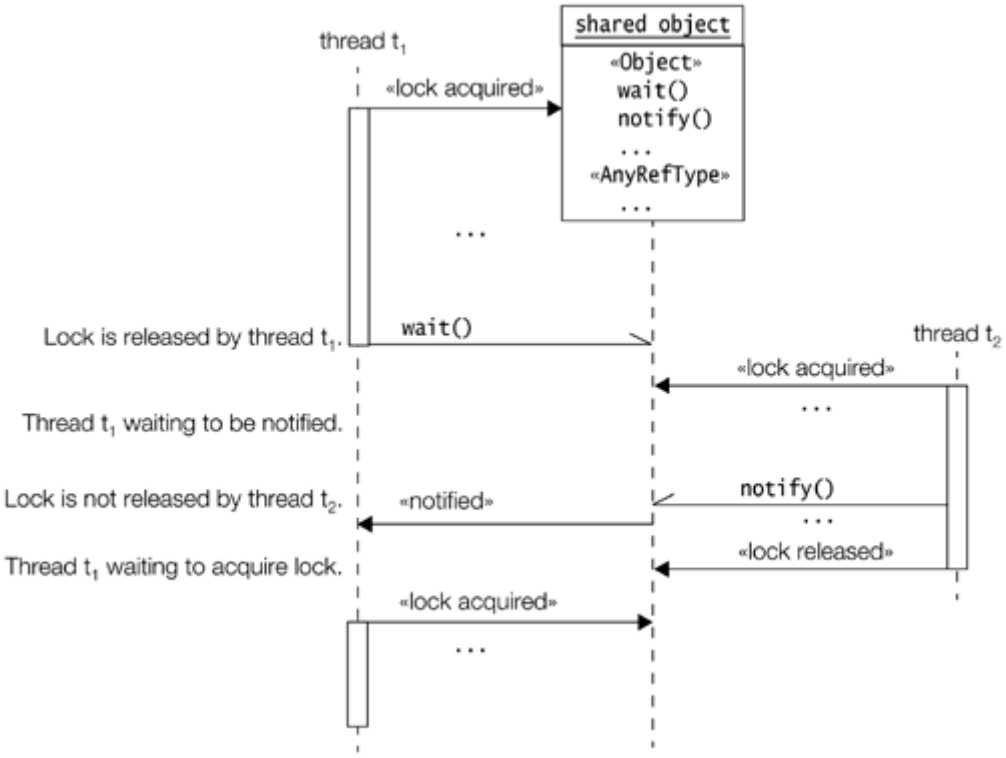
Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one atomic (non-interruptable) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.

Note that the waiting thread does not relinquish any other object locks that it might hold, only that of the object on which the wait() method was invoked. Objects that have these other locks remain locked while the thread is waiting.

Each object has a wait set containing threads waiting for notification. Threads in the Waiting-for-notification state are grouped according to the object whose wait() method they invoked.

Figure below shows a thread t1 that first acquires a lock on the shared object, and afterwards invokes the wait() method on the shared object. This relinquishes the object lock and the thread t1 awaits to be notified. While the thread t1 is waiting, another thread t2 can acquire the lock on the

shared object for its own purpose.



A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.

2. The waiting thread times out.

3. Another thread interrupts the waiting thread.

Notify:

Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM. On being notified, a waiting thread first transits to the Blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready-to-run state. The thread is also removed from the wait set of the object. Note that the object lock is not relinquished when the notifying thread invokes the notify() method. The notifying thread relinquishes the lock at its own discretion, and the awakened thread will not be able to run until the notifying thread relinquishes the object lock.

When the notified thread obtains the object lock, it is enabled for execution, waiting in the Ready-to-run state for its turn to execute again. Finally, when it does get to execute, the call to the wait() method returns and the thread can continue with its execution.

A call to the notify() method has no consequences if there are no threads in the wait set of the object.

In contrast to the notify() method, the notifyAll() method wakes up all threads in the wait set of the shared object. They will all transit to the Blocked-for-lock-acquisition state and contend for the object lock as explained earlier.

Time-out

The wait() call specified the time the thread should wait before being timed out, if it was not awakened as explained earlier. The awakened thread competes in the usual manner to execute again. Note that the awakened thread has no way of knowing whether it was timed out or awakened by one of the notification methods.

Interrupt

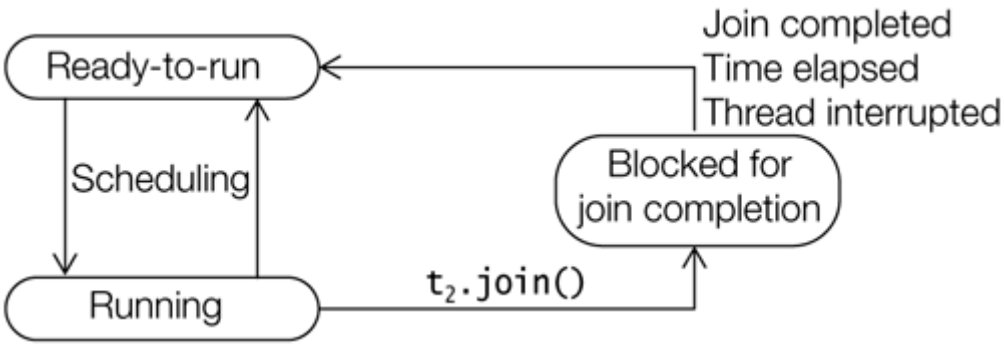
This means that another thread invoked the interrupt() method on the waiting thread. The awakened thread is enabled as previously explained, but if and when the awakened thread finally gets a chance to run, the return from the wait() call will result in an InterruptedException. This is the reason why the code invoking the wait() method must be prepared to handle this checked exception.

Joining

A thread can invoke the overloaded method join() on another thread in order to wait for the other thread to complete its execution before continuing, that is, the first thread waits for the second thread to join it after completion. A running thread t1 invokes the method join() on a thread t2. The join() call has no effect if thread t2 has already completed. If thread t2 is still alive, then thread t1 transits to the Blocked-for-join-completion state. Thread t1 waits in this state until one of these events occur:

- **Thread t2 completes:** In this case thread t1 is enabled and when it gets to run, it will continue normally after the join() method call.
- **Thread t1 is timed out:** The time specified in the argument in the join() method call has elapsed, without thread t2 completing. In this case as well, thread t1 is enabled. When it gets to run, it will continue normally after the join() method call.
- **Thread t1 is interrupted:** Some thread interrupted thread t1 while thread t1 was waiting for join completion. Thread t1 is enabled, but when it gets to execute, it will now throw an InterruptedException.

Figure: Joining of Threads



Example below illustrates joining of threads. The AnotherClient class below uses the Counter class, which extends the Thread class from Example 9.2. It creates two threads that are enabled for execution. The main thread invokes the join() method on the Counter A thread. If the Counter A thread has not already completed, the main thread transits to the Blocked-for-join-completion state. When the Counter A thread completes, the main thread will be enabled for running. Once the main thread is running, it continues with execution after (5). A parent thread can call the isAlive() method to find out whether its child threads are alive, before terminating itself. The call to the isAlive() method on the Counter A thread at (6) correctly reports that the Counter A thread is not alive. A similar scenario transpires between the main thread and the Counter B thread. The main thread passes through the Blocked-for-join-completion state twice at the most.

Example: Joining of Threads

```
class Counter extends Thread {

    private int currentValue;

    public Counter(String threadName) {
        super(threadName);                          // (1) Initialize thread.
        currentValue = 0;
        System.out.println(this);
        start();                                    // (2) Start this thread.
    }

    public int getValue() { return currentValue; }
    public void run() {                             // (3) Override from
superclass.
        try {
            while (currentValue < 5) {
                System.out.println(getName() + ": " + (currentValue++));
                Thread.sleep(250);                  // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + getName());
    }
}

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

        Counter counterA = new Counter("Counter A");
        Counter counterB = new Counter("Counter B");

        try {
            System.out.println("Wait for the child threads to finish.");
            counterA.join();                                    // (5)
            if (!counterA.isAlive())                            // (6)
```

```
                System.out.println("Counter A not alive.");
            counterB.join();                                  // (7)
            if (!counterB.isAlive())                          // (8)
                System.out.println("Counter B not alive.");
        } catch (InterruptedException e) {
            System.out.println("Main Thread interrupted.");
        }
        System.out.println("Exit from Main Thread.");
    }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Wait for the child threads to finish.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from Counter A.
Counter A not alive.
Exit from Counter B.
Counter B not alive.
Exit from Main Thread.
```

Blocking for I/O

A running thread, on executing a blocking operation requiring a resource (like a call to an I/O method), will transit to the Blocked-for-I/O state. The blocking operation must complete before the thread can proceed to the Ready-to-run state. An example is a thread reading from the standard input terminal, which blocks until input is provided:

```
int input = System.in.read();
```

Thread Termination

A thread can transit to the Dead state from the Running or the Ready-to-run states. The thread dies when it completes its run() method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected. There is no way the thread can be enabled for running again, not even by calling the start() method once more on the thread object.

Example below illustrates a typical scenario where a thread can be controlled by one or more threads.

```
class Worker implements Runnable {
    private Thread theThread;

    public void kickStart() {
        if (theThread == null) {
            theThread = new Thread(this);
            theThread.start();
        }
    }
```

```
    public void terminate() {
        theThread = null;
    }

    public void run() {
        while (theThread == Thread.currentThread()) {
            System.out.println("Going around in loops.");
        }
    }
}

public class Controller {
    public static void main(String[] args) {
        Worker worker = new Worker();
        worker.kickStart();
        Thread.yield();
        worker.terminate();
    }
}
```

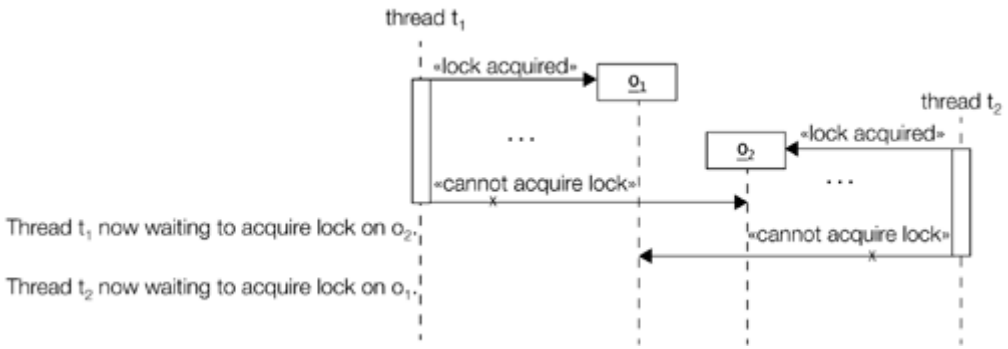
Possible output from the program:

```
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
```

Deadlocks

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be deadlocked.

A deadlock is depicted in Figure below. Thread t1 has a lock on object o1, but cannot acquire the lock on object o2. Thread t2 has a lock on object o2, but cannot acquire the lock on object o1. They can only proceed if one of them relinquishes a lock the other one wants, which is never going to happen.



1. **A demon Thread group**

- **can have non demon threads.**
- **does not exist after all the threads in the group have finished executing.**

2. **fgg**

- **The runtime environment distinguishes between user threads and daemon threads.**
- **As long as a user thread is alive, the JVM does not terminate.**
- **A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program.**
- **Daemon threads exist only to serve user threads.**

When a standalone application is run, a user thread is automatically created to execute the main() method. This thread is called the main thread. If no other user threads are spawned, the program terminates when the main() method finishes executing.

All other threads, called child threads, are spawned from the main thread, inheriting its user-thread status.

The main() method can then finish, but the program will keep running until all the user threads have finished.

Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.

Any attempt to change the status after the thread has been started, throws an IllegalThreadStateException.

Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This user thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have died after the main() method finished executing.

- **implementing the java.lang.Runnable interface**
- **extending the java.lang.Thread class**

```
public interface Runnable {
    void run();
}
```

Constructor Summary

- **Thread**()
- **Thread**(String name)
- **Thread**(Runnable target)
- **Thread**(Runnable target, String name)
- **Thread**(ThreadGroup group, Runnable target)
- **Thread**(ThreadGroup group, Runnable target, String name)
- **Thread**(ThreadGroup group, Runnable target, String name, long stackSize)
- **Thread**(ThreadGroup group, String name)

The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run () method that will be executed by the thread. An object of this class is a Runnable object.

2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.

3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately

after a thread has been spawned.

Example

```
class Counter implements Runnable {

    private int currentValue;

    private Thread worker;

    public Counter(String threadName) {
        currentValue = 0;
        worker = new Thread(this, threadName);   // (1) Create a new thread.
        System.out.println(worker);
        worker.start();                          // (2) Start the thread.
    }

    public int getValue() { return currentValue; }

    public void run() {                          // (3) Thread entry point
        try {
            while (currentValue < 5) {
                System.out.println(worker.getName() + ": " + (currentValue++));
                Thread.sleep(250);               // (4) Current thread sleeps.
            }
        } catch (InterruptedException e) {
            System.out.println(worker.getName() + " interrupted.");
        }
        System.out.println("Exit from thread: " + worker.getName());
    }
}

public class Client {
    public static void main(String[] args) {
        Counter counterA = new Counter("Counter A"); // (5) Create a thread.

        try {
            int val;
            do {
                val = counterA.getValue();       // (6) Access the counter
value.
                System.out.println("Counter value read by main thread: " +
val);
                Thread.sleep(1000);              // (7) Current thread sleeps.
            } while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
        }

        System.out.println("Exit from main() method.");
    }
}
```

void start()

This method spawns a new thread, that is, the new thread will begin execution as a child thread of

the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an IllegalThreadStateException if the thread was already started.

.

`void` **`checkAccess`**`()`
Determines if the currently running thread has permission to modify this thread.

`void` **`destroy`**`()`
Destroys this thread, without any cleanup.

`ClassLoader` **`getContextClassLoader`**`()`
Returns the context ClassLoader for this Thread.

`String` **`getName`**`()`
Returns this thread's name.

`int` **`getPriority`**`()`
Returns this thread's priority.

`ThreadGroup` **`getThreadGroup`**`()`
Returns the thread group to which this thread belongs.
`void` **`interrupt`**`()`
Interrupts this thread. `boolean`

`isAlive``()`
Tests if this thread is alive.

`boolean` **`isDaemon`**`()`
Tests if this thread is a daemon thread.

`boolean` **`isInterrupted`**`()`
Tests whether this thread has been interrupted.

`void` **`join`**`()`
Waits for this thread to die.

`void` **`join`**`(long millis)`
Waits at most `millis` milliseconds for this thread to die.

`void` **`join`**`(long millis, int nanos)`
Waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.

`void` **`resume`**`()`
Deprecated. *This method exists solely for use with* *`suspend()`*, *which has been deprecated*

because it is deadlock-prone. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.

`void` **`run`**`()`
If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

`void` **`setContextClassLoader`**`(ClassLoader cl)`
Sets the context ClassLoader for this Thread.

`void` **`setDaemon`**`(boolean on)`
Marks this thread as either a daemon thread or a user thread.

`void` **`setName`**`(String name)`
Changes the name of this thread to be equal to the argument `name`.

`void` **`setPriority`**`(int newPriority)`

Changes the priority of this thread.

`void` **`start`**`()`
Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread

Throws: IllegalThreadStateException - if the thread was already started.

`String` **`toString`**`()`
Returns a string representation of this thread, including the thread's name, priority, and thread group. .

`void` **`stop`**`()`
Deprecated. *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*
`void` **`stop`**`(Throwable obj)`
Deprecated. *This method is inherently unsafe. See `stop()` (with no arguments) for details. An additional danger of this method is that it may be used to generate exceptions that the target thread is unprepared to handle (including checked exceptions that the thread could not possibly throw, were it not for this method). For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.* `void` **`suspend`**`()`
Deprecated. *This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume`, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.*

List of static methods

`static int` **`activeCount`**`()`
Returns the number of active threads in the current thread's thread group.

`static void` **`dumpStack`**`()`
Prints a stack trace of the current thread.

`static int` **`enumerate`**`(Thread[] tarray)`
Copies into the specified array every active thread in the current thread's thread group and its subgroups.

`static Thread` **`currentThread`**`()`
Returns a reference to the currently executing thread object.

`static boolean` **`holdsLock`**`(Object obj)`

Returns `true` if and only if the current thread holds the monitor lock on the specified object.

`static boolean` **`interrupted`**`()`
Tests whether the current thread has been interrupted.

`static void` **`sleep`**`(long millis)`
Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
`throws InterruptedException`

`static void` **`sleep`**`(long millis, int nanos)`
Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds

Throws: IllegalArgumentException - if the value of millis is negative or the value of nanos is not in the range 0-999999. InterruptedException - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

`static void` **`yield`**`()`
Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Synchronized Blocks

Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object. The general form of the synchronized statement is as follows:

synchronized (<**object reference expression**>) { <code block> }

The <object reference expression> must evaluate to a non-null reference value, otherwise, a NullPointerException is thrown.

The code block is usually related to the object on which the synchronization is being done. This is the case with synchronized methods, where the execution of the method is synchronized on the lock of the current object:

Object specification in the synchronized statement is mandatory.

A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block.

The braces of the block cannot be left out, even if the code block has just one statement.

```
public Object pop() {
    synchronized (this) {           // Synchronized block on current object
        // ...
    }
}
```

1. **Classes also have a class-specific lock that is analogous to the object lock.**
2. **Such a lock is actually a lock on the java.lang.Class object associated with the class.**
3. **Given a class A, the reference A.class denotes this unique Class object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.**

Synchronized blocks can also be specified on a class lock:

synchronized (<class name>.class) { <code block> }

The block synchronizes on the lock of the object denoted by the reference <class name>.class. A static synchronized method classAction() in class A is equivalent to the following declaration:

```
static void classAction() {
    synchronized (A.class) {       // Synchronized block on class A
        // ...
    }
}
```

In summary, a thread can hold a lock on an object

- **by executing a synchronized instance method of the object**
- **by executing the body of a synchronized block that synchronizes on the object**
- **by executing a synchronized static method of a class**

Question 6

```
class A extends Thread {
  String[] sa;
  public A(String[] sa) {this.sa = sa;}
  public void run() {
    synchronized (sa) {
      System.out.print(sa[0] + sa[1] + sa[2]);
}}}
class B {
  private static String[] sa = new String[]{"X","Y","Z"};
  public static void main (String[] args) {
    synchronized (sa) {
      Thread t1 = new A(sa);
      t1.start();
      sa[0] = "A"; sa[1] = "B"; sa[2] = "C";
}}}
```

explanation

The block inside the main method is synchronized on the String array object sa. Inside the block, a new thread t1 is started and will run at the discretion of the thread scheduler. The A.run method also contains a block that is synchronized on the String array object sa. Even if the thread scheduler moves thread t1 into the Running state, it will block while attempting to acquire the lock of the String array object sa. Thread t1 will continue to block until the synchronized block in the B.main method runs to completion. At that time, the contents of the String array object have all been updated.

What is the result of attempting to compile and run the program?

a. Prints: XYZ

b. Prints: AYZ

c. Prints: ABZ

d. Prints: ABC<<<<<<

e. Compile-time error

f. Run-time error

g. None of the above

Question 5

```
class A extends Thread {
  public void run() {
    synchronized (this) {
      try {wait(5000);} catch (InterruptedException ie){}
  }}
  public static void main(String[] args) {
    A a1 = new A();
    long startTime = System.currentTimeMillis();
    a1.start();
    System.out.print(System.currentTimeMillis() - startTime + ",");
    try {a1.join(6000);} catch (InterruptedException ie) {}
    System.out.print(System.currentTimeMillis() - startTime);
}}
```

What are the possible results of attempting to compile and run the program?

a. >>>>>>	The first number printed is greater than or equal to 0
b.	The first number printed must always be greater than 5000
c. .>>>>>>>>>	The second number printed must always be greater than 5000
d.	The second number printed must always be greater than 6000
e.	The synchronized block inside the run method is not necessary
f.	Compile-time error
g.	Run-time error

Question 11

Which of the following thread state transitions model the lifecycle of a thread?

a.	The Dead state to the Ready state
b.	The Ready state to the Not-Runnable state
c.	The Ready state to the Running state
d.	The Running state to the Not-Runnable state
e.	The Running state to the Ready state
f.	The Not-Runnable state to the Ready state
g.	The Not-Runnable state to the Running state

c d e f

Question 9

```
synchronized (expression) block
```

b e f g h i

The synchronized statement has the form shown above. Which of the following are true statements?

a.	A compile-time error occurs if the expression produces a value of any reference type
b.	A compile-time error occurs if the expression produces a value of any primitive type
c.	A compile-time error does not occur if the expression is of type boolean
d.	The sychronized block may be processed normally if the expression is null
e.	If execution of the block completes normally, then the lock is released
f.	If execution of the block completes abruptly, then the lock is released

g.	A thread can hold more than one lock at a time
h.	Synchronized statements can be nested
i.	Synchronized statements with identical expressions can be nested

A thread inherits its daemon status from the thread that created it

For the purposes of the exam, invoking the start method on a thread that has already been started will generate an IllegalThreadStateException. The actual behavior of Java might be different. If the start method is invoked on a thread that is already running, then an IllegalThreadStateException will probably be thrown. However, if the thread is already dead then the second attempt to start the thread will probably be ignored, and no exception will be thrown. However, for the purposes of the exam, the exception is always thrown in response to the second invocation of the start method. This is a case where the exam tests your knowledge of the specification of the Thread.start method and ignores the actual behavior of the 1.4 version of the JVM.

Waiting and Notifying

Waiting and notifying provide **means of communication between threads** that synchronize on the same object (see Section 9.4, p. 359).
The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose.
These final methods are defined in the Object class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException.

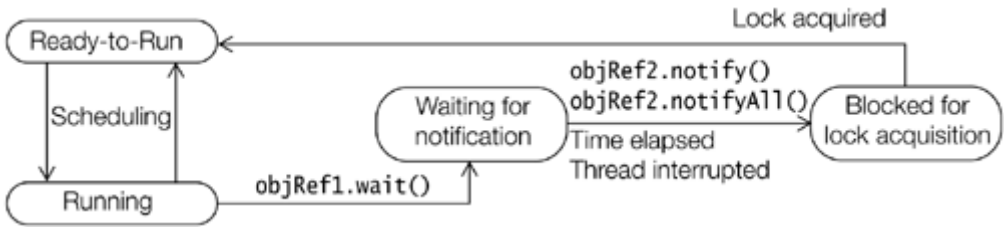
```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

A thread invokes the **wait()** method on the object whose lock it holds. The thread is added to the wait set of the object

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

Figure 9.6. Waiting and Notifying



Threads

1. **The runtime environment distinguishes between user threads and daemon threads.**
2. **As long as a user thread is alive, the JVM does not terminate.**
3. **A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program.**
4. **Daemon threads exist only to serve user threads.**
5. When a standalone application runs a user thread is automatically created to execute the main() method. This thread is called the main thread. If no other user threads are spawned, the program terminates when the main() method finishes executing.
6. **All other threads, called child threads, arNoTes - e spawned from the main thread, inheriting its user-thread status.**
7. **The main() method can then finish, but the program will keep running until all the user threads have finished.**
8. **Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.**
9. **Any attempt to change the status after the thread has been started, throws an IllegalThreadStateException.**
10. Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.
11. The life of the Daemon thread is specified by the application. If all the user threads in the application perish, JVM stops it.
12. Threads can be created using 2 ways
    implementing the java.lang.Runnable interface
    extending the java.lang.Thread class
13. Constructor Summary
    - **Thread**()
    - **Thread**(String name)
    - **Thread**(Runnable target)
    - **Thread**(Runnable target, String name)
    - **Thread**(ThreadGroup group, String name)
    - **Thread**(ThreadGroup group, Runnable target)
    - **Thread**(ThreadGroup group, Runnable target, String name)
    - **Thread**(ThreadGroup group, Runnable target, String name, long stackSize)
14. List of static methods
    - static int **activeCount**()
      Returns the number of active threads in the current thread's thread group.
    - static int **enumerate**(Thread[] tarray)
      Copies into the specified array every active thread in the current thread's thread group and its subgroups.
    - static void **dumpStack**()
      Prints a stack trace of the current thread.
    - static Thread **currentThread**()
      Returns a reference to the currently executing thread object.
    - static boolean **holdsLock**(Object obj)
      Returns true if and only if the current thread holds the monitor lock on the specified object.
    - static boolean **interrupted**()
      Tests whether the current thread has been interrupted.
    - static void **sleep**(long millis)
      Causes the currently executing thread to sleep (temporarily cease execution)

for the specified number of milliseconds.
- § IllegalArgumentException
- § throws InterruptedException

- o static void **sleep**(long millis, int nanos)
  Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds
  - § **Throws:**
    - IllegalArgumentException - if the value of millis is negative or the value of nanos is not in the range 0-999999.
    - InterruptedException - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.
- o static void **yield**()
  Causes the currently executing thread object to temporarily pause and allow other threads to execute.

15. **Synchronized Blocks**
    - **o** execution of synchronized methods of an object is synchronized on the lock of the object,
    - **o** the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.
16. The general form of the synchronized statement is as follows:
    - o synchronized (<**object reference expression**>) { <code block>
    - o **The <object reference expression> must evaluate to a** non-null **reference value, otherwise, a NullPointerException is thrown.**
17. **Object specification in the synchronized statement is mandatory.**
18. **The braces of the block cannot be left out, even if the code block has just one statement.**
    synchronized (this) **{** // Synchronized block on current object **}**
19. Classes also have a **class-specific lock.**
    - o lock on the **java.lang.Class object** associated with the class**.**
    - o the reference **A.class** denotes this unique Class object.
    - o The class lock can be used in much the same way as an object lock to implement mutual exclusion.
20. Synchronized blocks can also be specified on a class lock:
    synchronized (<class name>.class) { <code block> }
    - o The block synchronizes on the lock of the object denoted by the reference <class name>.class.
    - o A static synchronized method classAction() in class A is equivalent to the following declaration:
      static void classAction() {
        synchronized (A.class) { // Synchronized block on class A }
      }
21. A thread can hold a lock on an object by
    - o executing a synchronized instance method of the object
    - o by executing a synchronized static method of a class.
    - o by executing the body of a synchronized block that synchronizes on the object
22. life cycle of a thread
    - o **Ready-to-run state**: thread starts life in the Ready-to-run state.
    - o **Running state**: If a thread is in the Running state, it means that the thread is currently executing
    - o **Dead state**: Once in this state, the thread cannot ever run again
    - o **Non-runnable states**

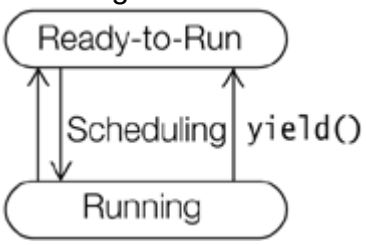
A running thread can transit to one of the non-runnable states, depending on the circumstances.
A thread remains in a non-runnable state until a special transition occurs.
A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.
The non-runnable states can be characterized as follows:

- **Sleeping:** The thread sleeps for a specified amount of time
- **Blocked for I/O**: The thread waits for a blocking operation to complete
- **Blocked for join completion**: The thread awaits completion of another thread
- **Waiting for notification**: The thread awaits notification from another thread
- **Blocked for lock acquisition**: The thread waits to acquire the lock of an object.

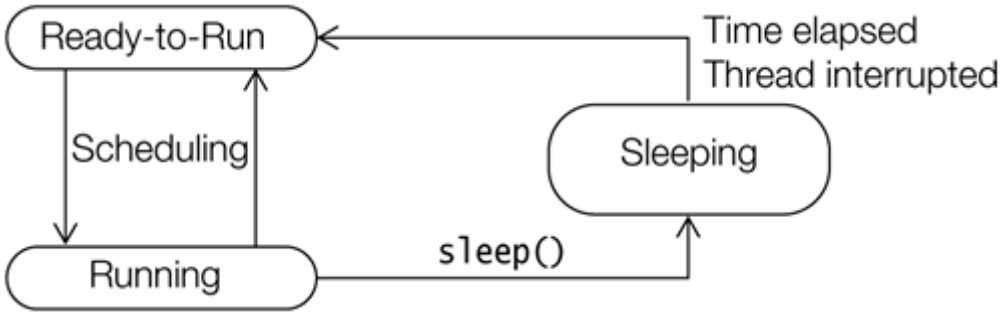
23. Thread Priority:

- final int getPriority()
  final void setPriority(int newPriority)
- The priority set will be the minimum of the two values: the specified newPriority and the maximum priority permitted for this thread.
- Priorities are integer values from
  - **1** (lowest priority given by the constant **Thread. MIN_PRIORITY**) to
  - **10** (highest priority given by the constant **Thread.MAX_PRIORITY**). The default priority is **5** (**Thread.NORM_PRIORITY**).

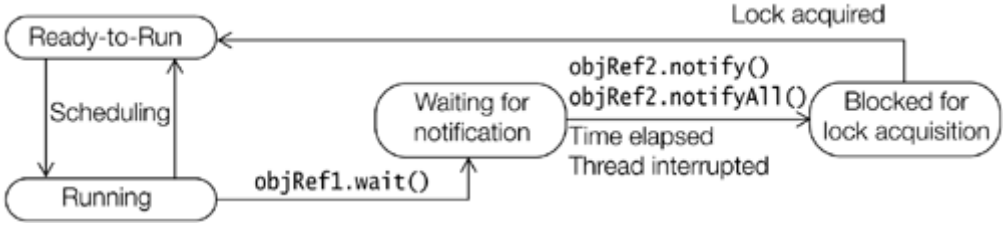
24. Thread.yield(): Running and Yielding >>no release of lock



25. Thread.Sleep(): Sleeping and waking up >>no release of lock



26. objLock.Wait(): Waiting and Notifying



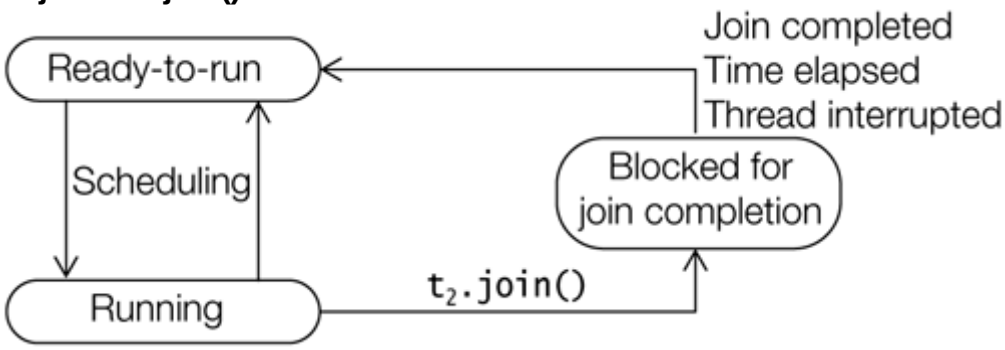
- Waiting and notifying provide means of **communication between threads that synchronize on the same object**
- The threads execute wait() and notify() (or notifyAll()) methods on the shared

object for this purpose.

- Wait(), notify & notifyall() r final methods defined in the Object class, and therefore, inherited by all objects.
- A thread invokes the **wait()** method on the object whose lock it holds. The thread is added to the wait set of the object
- **These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException**
- final void wait(long timeout) throws **InterruptedException <<try catch neccasry**
- final void wait(long timeout, int nanos) throws InterruptedException
- final void wait() throws InterruptedException

- A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.
- thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:
  - Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
  - The waiting thread times out.
  - Another thread interrupts the waiting thread.

27. **objThread.join(): JOINING**



- A running thread t_1 invokes the method join() on a thread t_2.
- The join() call has no effect if thread t_2 has already completed.
- If thread t_2 is still alive, then thread t_1 transits to the Blocked-for-join-completion state.
- Thread t_1 waits in this state until one of these events occur:
  - Thread t_2 completes.
    - In this case thread t_1 is enabled and when it gets to run, it will continue normally after the join() method call.
  - Thread t_1 is timed out. The time specified in the argument in the join() method call has elapsed, without thread t_2 completing.
    - thread t_1 is enabled. When it gets to run, it will continue normally after the join() method call.
  - Thread t_1 is interrupted.
    Some thread interrupted thread t_1 while thread t_1 was waiting for join completion.
    - Thread t_1 is enabled, but when it gets to execute, it will now throw an InterruptedException.

28. **ThreadGroup Class**
- The ThreadGroup class manages groups of threads for Java applications. A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- **Uncaught exceptions for thread are handled by ThreadGroup. uncaughtException()**
- Constructor Summary
  - **ThreadGroup**(String name):: Constructs a new thread group.
  - **ThreadGroup**(ThreadGroup parent, String name) Creates a new thread group.

ThreadGroups can contain not only threads but also other ThreadGroups. The top most

thread group in a Java application is the thread group named "main". You can create threads and thread groups in the "main" group. You can also create threads and thread groups in subgroups of "main" and so on. The result is a root-like hierarchy of threads and thread groups.

Identifiers: In Java an identifier is composed of a sequence of characters, where each character can be either a letter, a digit, a connecting punctuation (such as underscore _), or any currency symbol (such as $, ¢, ¥, or £). However, the first character in an identifier cannot be a digit.

Identifiers in Java are case sensitive

Examples of Legal Identifiers:
```
number, Number, sum_$, bingo, $$_100, mål, grüß
```

Examples of Illegal Identifiers:
```
48chevy, all@hands, grand-sum
```

Table 2.1. Keywords in Java

abstract	default	implements	protected	throw
assert	do	import	public	throws
boolean	double	instanceof	return	transient
break	else	int	short	try
byte	extends	interface	static	void
case	final	long	**strictfp**	volatile
catch	finally	native	super	while
char	float	new	switch	
class	for	package	synchronized	
continue	if	private	this	

Table 2.2. Reserved Literals in Java

null	true	false

Table 2.3. Reserved Keywords not Currently in Use

const	goto

Integer Literals

Integer data types are comprised of the following primitive data types: int, long, byte, and short

The default data type of an integer literal is always int, but it can be specified as long by appending the suffix L (or l) to the integer value. Without the suffix, the long literals 2000L and 0l will be

interpreted as int literals. There is no direct way to specify a short or a byte literal.

In addition to the decimal number system, integer literals can also be specified in octal (base 8) and hexadecimal (base 16) number systems. Octal and hexadecimal numbers are specified with 0 and 0x (or 0X) prefix respectively.

Negative integers (e.g. -90) can be specified by prefixing the minus sign (-) to the magnitude of the integer regardless of number system (e.g., -0132 or -0X5A).

Java does not support literals in binary notation.

Table 2.5. Examples of Decimal, Octal, and Hexadecimal Literals

Decimal	Octal	Hexadecimal
8	010	0x8
10L	012L	0XaL
16	020	0x10
27	033	0x1B
90L	0132L	0x5aL
-90	-0132	-0X5A
2147483647 (i.e., 231-1)	017777777777	0x7fffffff
-2147483648 (i.e., -231)	-020000000000	-0x80000000
1125899906842624L (i.e., 250)	040000000000000000L	0x4000000000000L

Floating-point Literals

Floating-point data types come in two flavors: **float** or **double**.

The default data type of a floating-point literal is double, but it can be explicitly designated by appending the suffix D (or d) to the value. A floating-point literal can also be specified to be a float by appending the suffix F (or f).

Floating-point literals can also be specified in **scientific notation**, where E (or e) stands for Exponent. For example, the double literal 194.9E-2 in scientific notation is interpreted as 194.9*10-2 (i.e., 1.949).

Examples of double Literals

```
0.0       0.0d       0D
0.49      .49        .49D
49.0      49.        49D
4.9E+1    4.9E+1D    4.9e1d   4900e-2   .49E2
```

Examples of float Literals

```
0.0F      0f
0.49F     .49F
49.0F     49.F       49F
4.9E+1F   4900e-2f   .49E2F
```

Note that the decimal point and the exponent are optional and that at least one digit must be specified.

Character Literals

A character literal is quoted in single-quotes ('). All character literals have the primitive data type char.

Characters in Java are represented by the 16-bit Unicode character set, which subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. Any Unicode character can be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix \u.

Table 2.6. Examples of Unicode Values

Character Literal	Character Literal using Unicode value	Character
' '	'\u0020'	Space
'0'	'\u0030'	0
'1'	'\u0031'	1
'9'	'\u0039'	9
'A'	'\u0041'	A
'B'	'\u0042'	B
'Z'	'\u005a'	Z
'a'	'\u0061'	a
'b'	'\u0062'	b

'z'	'\u007a'	z

Escape Sequences

These escape sequences can be single-quoted to define character literals. For example, the character literals '\t' and '\u0009' are equivalent.

Table 2.7. Escape Sequences

Escape Sequence	Unicode Value	Character
\b	\u0008	Backspace (BS)
\t	\u0009	Horizontal tab (HT or TAB)
\n	\u000a	Linefeed (LF) a.k.a., Newline (NL)
\f	\u000c	Form feed (FF)
\r	\u000d	Carriage return (CR)
\'	\u0027	Apostrophe-quote
\"	\u0022	Quotation mark
\\	\u005c	Backslash

We can also use the escape sequence \ddd to specify a character literal by octal value, where each digit d can be any octal digit (0–7), as shown in Table 2.8. The number of digits must be three or fewer, and the octal value cannot exceed \377, that is, only the first 256 characters can be specified with this notation.

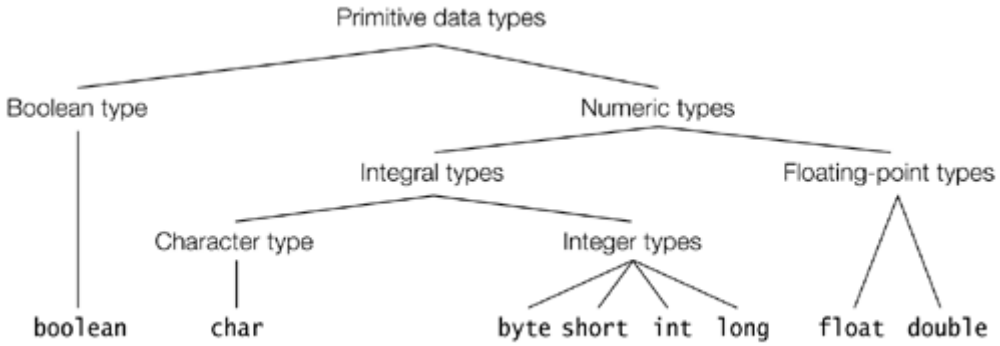
Table 2.8. Examples of Escape Sequence \ddd

Escape Sequence \ddd	Character Literal
'\141'	'a'
'\46'	'&'
'\60'	'0'

Comments

Java provides three types of comments to document a program:

- A single-line comment: // ... to the end of the line
- A multiple-line comment: /* ... */
- A documentation (Javadoc) comment: /** ... */



Primitive data types in Java can be divided into three main categories:

- Integral types— represent signed integers (**byte, short, int, long**) and unsigned character values (**char**)
- Floating-point types (**float, double**)— represent fractional signed numbers
- Boolean type (**boolean**)— represent logical values

Each primitive data type also has a corresponding wrapper class that can be used to represent a primitive value as an object.

Table 2.10. Range of Character Values

Data Type	Width (bits)	Minimum Unicode value	Maximum Unicode value
char	16	0x0 (\u0000)	0xffff (\uffff)

Characters are represented by the data type char. Their values are unsigned integers that denote all the 65536 (216) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters.

The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

Floating-point Types

Zero can be either 0.0 or -0.0.

Table 2.13. Summary of Primitive Data Types

Data Type	Width (bits)	Minimum Value, Maximum Value	Wrapper Class
boolean	not applicable	true, false (no ordering implied)	Boolean
byte	8	-27, 27-1	Byte
short	16	-215, 215-1	Short
char	16	0x0, 0xffff	Character
int	32	-231, 231-1	Integer
long	64	-263, 263-1	Long
float	32	±1.40129846432481707e-45f, ±3.40282347663852886e+38f	Float
double	64	\'b14.94065645841246544e-324, \'b11.79769313486231570e+308	Double

Default values for fields of primitive data types and reference types. The value assigned depends on the type of the field.

Table 2.14. Default Values

Data Type	Default Value
boolean	false
char	'\u0000'
Integer (byte, short, int, long)	0L for long, 0 for others
Floating-point (float, double)	0.0F or 0.0D
Reference types	null

If no initialization is provided for a static variable either in the declaration or in a static initializer block, it is initialized with the default value of its type when the class is loaded.

Similarly, if no initialization is provided for an instance variable either in the declaration or in an instance initializer block, it is initialized with the default value of its type when the class is instantiated.

The fields of reference types are always initialized with the null reference value, if no initialization is provided.

Note that static variables are initialized when the class is loaded the first time, and instance variables are initialized accordingly in every object created from the class Light.

Initializing Local Variables of Primitive Data Types

Local variables are not initialized when they are created at method invocation, that is, when the execution of a method is started. They must be explicitly initialized before being used. The compiler will report attempts to use uninitialized local variables.

Example 2.2 Flagging Uninitialized Local Variables of Primitive Data Types

```
public class TooSmartClass {
    public static void main(String[] args) {
        int weight = 10, thePrice;                          // Local variables

        if (weight <  10) thePrice = 1000;
        if (weight >  50) thePrice = 5000;
        if (weight >= 10) thePrice = weight*10;             // Always executed.
        System.out.println("The price is: " + thePrice);   // (1)
    }
}
```

The compiler complains that the local variable thePrice used in the println statement at (1) may not be initialized.

Initializing Local Reference Variables

Local reference variables are bound by the same initialization rules as local variables of primitive data types.

Example 2.3 Flagging Uninitialized Local Reference Variables

```
public class VerySmartClass {
    public static void main(String[] args) {
        String importantMessage;       // Local reference variable

        System.out.println("The message length is: " + importantMessage.length
());
    }
}
```

The compiler complains that the local variable importantMessage used in the println statement may not be initialized. If the variable importantMessage is set to the value null, the program will compile. However, at runtime, a NullPointerException will be thrown since the variable importantMessage will not denote any object. The golden rule is to ensure that a reference variable, whether local or not, is assigned a reference to an object before it is used, that is, ensure that it does not have the value null.

Which of the following lines are valid declarations?

Select the three correct answers.

a. char a = '\u0061';
b. char 'a' = 'a';
c. char \u0061 = 'a';
d. ch\u0061r a = 'a';
e. ch'a'r a = 'a';

Java Source File Structure

A Java source file can have the following elements that, if present, must be specified in the following order:

1. An **optional package declaration** to specify a package name.
2. **Zero or more import declarations**.
3. **Any number of top-level class and interface declarations**.

The classes and interfaces can be defined in any order. Class and interface declarations are collectively known as type declarations. Technically, a source file need not have any such

definitions, but that is hardly useful.

> The Java 2 SDK imposes the restriction that at the most one public class definition per source file can be defined. If a public class is defined, the file name must match this public class. If the public class name is NewApp, then the file name must be NewApp.java.

- Is an empty file a valid source file?
- Answer **true** or false.

The main() Method

The main() method must have public accessibility so that the interpreter can call it. It is a static method belonging to the class, so that no object of the class is required to start the execution. It does not return a value, that is, it is declared void. It always has an array of String objects as its only formal parameter. This array contains any arguments passed to the program on the command line

All this adds up to the following definition of the main() method:

```
public static void main(String[] args)
{
    // ...
}
```

- The main() method can also be overloaded like any other method
- The main() method can also be declared as final.
- The parameter in main can also be declared as final.

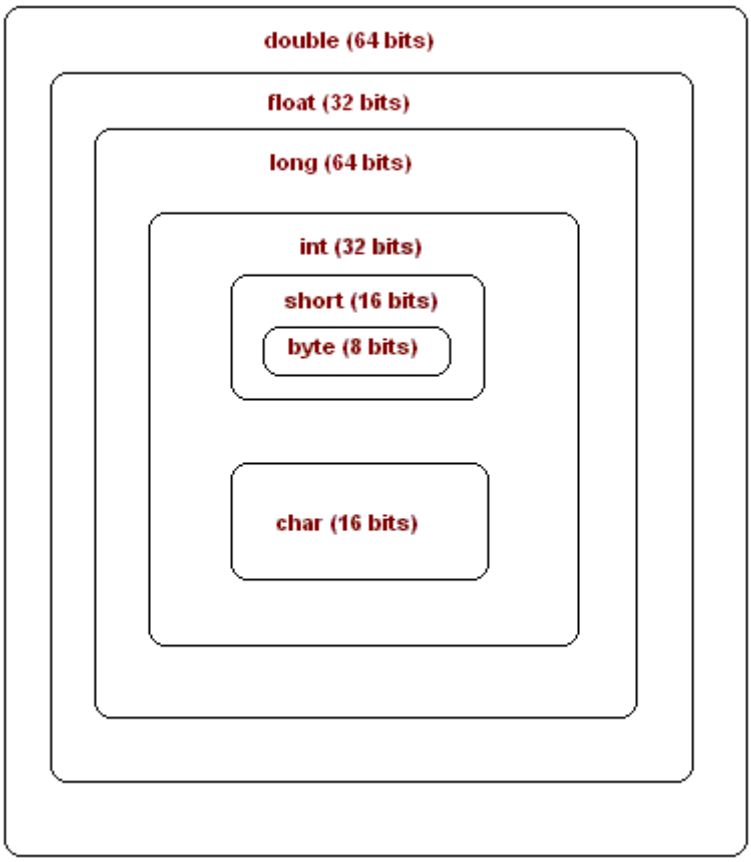
So this is also valid:

```
final public static void main(final String[] args)
{

}
```

1. Precedence Table:

Post	Postfix operators	[] . (parameters) expression++ expression--
Pre	Unary prefix operators	++expression --expression +expression -expression ~ !
CC @	Unary prefix creation and cast	new (type)
Mul-	Multiplicative	* / %
Add	Additive	+ -
Shift	Shift	<< >> >>>
Shows Rel~	Relational	< <= > >= instanceof
Equ~	Equality	== !=
Among Bitwise AND	Bitwise/logical AND	&
XOR	Bitwise/logical XOR	^
OR	Bitwise/logical OR	|
Cond~ AND	Conditional AND	&&
OR	Conditional OR	||
N cond~	Conditional	?:
Assi~	Assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=

2. Right to left Associatively: pre-conditional Ass
    a. Right associativity implies grouping from right to left:
    - - 4 is interpreted as (- (- 4))
3. Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.
4. All binary operators, except for the relational and assignment operators, associate from left to right.
5. The relational operators are nonassociative.
6. Precedence rules are used to determine which operator should be applied first if there are two operators with different precedence, and these follow each other in the expression. In such a case, the operator with the highest precedence is applied first.
7. Casting between primitive data types and reference types is not permitted.
8. Boolean values cannot be cast to other data values, and vice versa.
9. The reference literal null can be cast to any reference type.
10. different situations in which implicit conversions occur:
    a. Assignment conversion
    b. Method invocation conversion
    c. Numeric/Arithmetic promotion
    d. String conversion
11. **Rules for conversion of primitives in assignment**



a.
b. Non-boolean primitive data types can be converted to non-boolean only if it is widening conversion.

12. char is unsigned. So, short s = ch; //invalid, needs explicit casting as loss of information. Unsigned to Signed
13. operands of byte, short and char are promoted to int. binary/unary operator
    a. arry['a']
14. Binary numeric promotion implicitly applies appropriate widening primitive conversions so that a pair of operands have the broadest numeric type of the two, which is always at least int.
    a. If T is broader than int, both operands are converted to T; otherwise, both operands are converted to int.
15. Evaluation Order:
    a. the operands of operators are evaluated from left to right.
    b. <u>Evaluate Left-Hand Operand First</u>
    c. The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated.
    d. if the left-hand operand contains an assignment to a variable and the right-hand operand contains a reference to that same variable, then the value produced by the reference will reflect the fact that the assignment occurred first.
        i. int j = (i=3) * i; //9
    e. If the operator is a compound-assignment operator, then evaluation of the left-hand operand includes both remembering the variable that the left-hand operand denotes and fetching and saving that variable's value for use in the implied combining operation.
        i. int a = 9; a += (a = 3); // a=9+3
    f. If evaluation of the left-hand operand of a binary operator completes

abruptly, no part of the right-hand operand appears to have been evaluated.

i. int j = 1;
try {
int i = forgetIt() / (j = 2);
} catch (Exception e) {
System.out.println(e);
System.out.println("Now j = " + j);
}

_____
Prints: java.lang.Exception: exception thrown in forgetIt()
Now j = 1

g. Evaluate Operands before Operation

h. every operand of an operator (except the conditional operators &&, ||, and ? :) have to be fully evaluated before any part of the operation itself is performed.

i. int divisor = 0;
try {
int i = 1 / (divisor * loseBig());
} catch (Exception e) {
System.out.println(e);
}

_____
Print: java.lang.Exception: exception thrown in loseBig()
and not:
java.lang.ArithmeticException: / by zero

i. Argument Lists are Evaluated Left-to-Right

j. print3(s, s, s = "gone");

static void print3(String a, String b, String c) {
System.out.println(a + b + c);
}

_____
Print: going, going, gone

k. If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

l. Array Access Evaluation Order

m. the expression to the left of the brackets appears to be fully evaluated before any part of the expression within the brackets is evaluated

n. a[(a=b)[3]], the expression a is fully evaluated before the expression (a=b)[3]; this means that the original value of a is fetched and remembered while the expression (a=b)[3] is evaluated.

o. int[] a = { 11, 12, 13, 14 };
int[] b = { 0, 1, 2, 3 };
System.out.println(a[(a=b)[3]]);

_____
Print: 14

p. If evaluation of the expression to the left of the brackets completes abruptly, no part of the expression within the brackets will appear to have been evaluated.

q. If array reference expression produces null instead of a reference to an array, then a NullPointerException is thrown at run time, but only after all parts of the array access expression have been evaluated and only if these

evaluations completed normally.

```
int index = 1;
try {
    method()[index=2]++;
} catch (Exception e) {
    System.out.println(e + ", index=" + index);
}
```

_____

Print: java.lang.NullPointerException: thrown in method(), index=2

r. int a[] = null; int i = a[methodThrowingXexception()];

_____

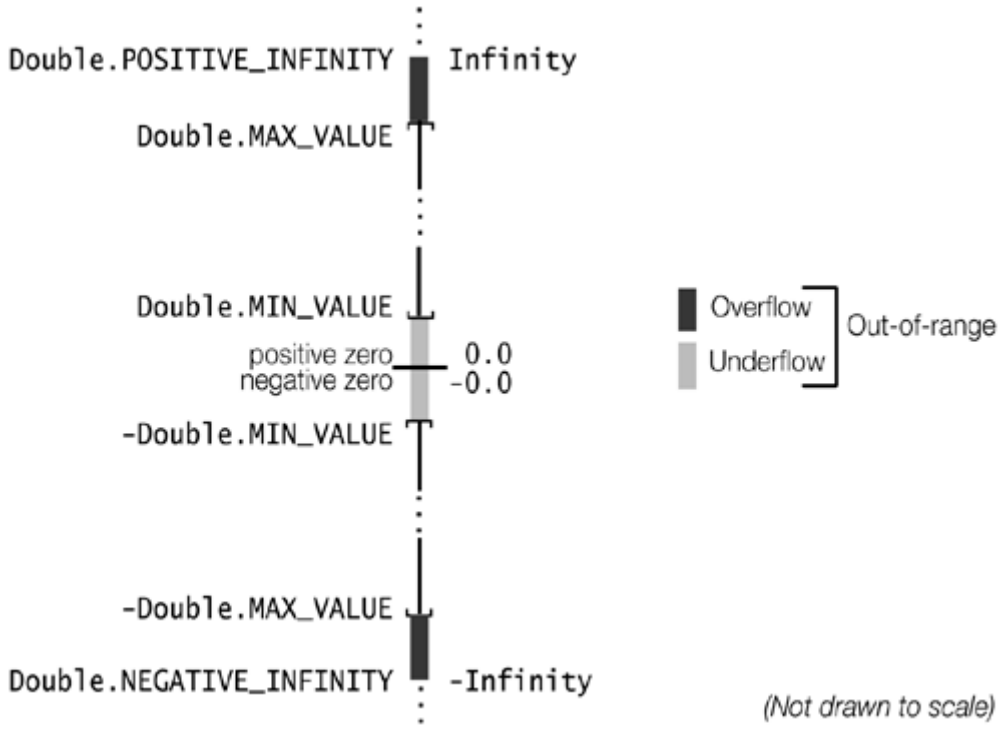
Print: ThrownXException Not NullPointerException

16. **Numeric Type Conversions on Assignment:**
    a widening primitive conversion is permissible, then the widening conversion is applied implicitly
    a. Integer values widened to floating-point values can result in loss of precision.
    b. float or double: The mantissa occupies the 23 least significant bits of a float and the 52 least significant bits of a double. The exponent, 8 bits in a float and 11 bits in a double, sits between the sign and mantissa.
    c. Integer has 32 bits => loss of precision
17. implicit narrowing primitive conversions on assignment can occur in cases where all of the following conditions are fulfilled:
    a. destination type is either byte, short, or char type
        i. source is a constant expression of either byte, short, char, or int type
        ii. value of the source is determined to be in the range of the destination type at compile time
    b. final int i1 = 20;
       byte b3 = i1;
    c. bye b = 128; //compilation error; need explicit casting (byte)
18. A compile-time error occurs if a decimal literal of type int (a literal w/o L suffixed) is larger than 2147483647 or less than -2147483648
19. **compile-time constant expression**
20. not all variables declared final (which is the general meaning of the term constants) are compile-time constant expressions.
    Requirements:
    a. The case labels in a switch statement must be constant expressions.
    b. The type of a conditional expression (the ?: operator) may be byte, short, or char if either the second or third operand "is of type T where T is byte, short, or char, and the other operand is a constant expressions of type int whose value is representable in type T." [JLS 15.25 Conditional Operator ? :]
    c. The implicit narrowing of integer constants in the assignment and method return conversion contexts requires that the value assigned or returned is a compile-time constant expression
    d. Inner classes can declare static fields , but only if they are declared final and initialized with the value of a compile-time constant expression (inlined constants).
    e. If the operands in a string concatenation operation are not compile-time constant expressions, the string concatenation operation must be executed at runtime.
    f. **A constant expression cannot include any value that could possibly**

change after compilation.

g. compile-time constant expressions are never variables, never invoke methods or constructors, and even more generally, never include reference types other than String.
h. ????References to inlined constants from other classes and interfaces do not cause the class or interface in which they are declared to be loaded, linked, and initialized. In other words, references to inlined constants does not constitute the "first active use" of a class or interface

21. **Range of Numeric Values**: MAX_VALUE and MIN_VALUE, which are defined in each numeric wrapper class
22. Integer Arithmetic
    a. Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which causes an ArithmeticException
    b. A valid value does not necessarily mean that the result is correct, as demonstrated by the following examples:
    ```
    int tooBig   = Integer.MAX_VALUE + 1;
       // -2147483648 which is Integer.MIN_VALUE.
    int tooSmall = Integer.MIN_VALUE - 1;
       //  2147483647 which is Integer.MAX_VALUE
    ```
    c. integer arithmetic wraps if the result is out-of-range
    d. In order to avoid wrapping of out-of-range values, programs should either use explicit checks or a wider type:
    ```
    long notTooBig   = Integer.MAX_VALUE + 1L;   //  2147483648L in range.
    long notTooSmall = Integer.MIN_VALUE - 1L;   // -2147483649L in range.
    ```
23. **<u>Floating-point Arithmetic</u>**
    a. floating-point operations result in values that are out-of-range.
    b. System.out.println(4.0 / 0.0); // Prints: Infinity
    c. System.out.println(-4.0 / 0.0); // Prints: -Infinity
    d.



    e. constants POSITIVE_INFINITY and NEGATIVE_INFINITY, in the wrapper classes java.lang.Float and java.lang.Double. A value can be compared with these constants to detect overflow.

f. Underflow: Floating-point arithmetic can also result in underflow to zero, that is, the value is too small to be represented as a double or float.
g. Underflow occurs in the following situations:
    i. +0.0 => result is between Double.MIN_VALUE (or Float. MIN_VALUE) and zero
    ii. -0.0 => result is between -Double.MIN_VALUE (or -Float. MIN_VALUE) and zero
h. Negative zero compares equal to positive zero, i.e. (-0.0 == 0.0) is true.
i. NaN: certain operations have no mathematical result
    i. square root of -1
    ii. System.out.println(0.0 / 0.0); // Prints: NaN
j. NaN is represented by the constant named NaN in the wrapper classes java.lang.Float and java.lang.Double
k. Any operation involving NaN produces NaN.
l. Any comparison (except inequality !=) involving NaN and any other value (including NaN) returns false.
m. inequality comparison of NaN with another value (including NaN) always returns true.
n. checking a value for NaN is to use the static method isNaN() defined in both wrapper classes, java.lang.Float and java.lang.Double.

24. Strict Floating-Point Arithmetic: strictfp modifier

a. To ensure identical results are produced on all JVMs, the keyword strictfp can be used to enforce strict behavior for floating-point arithmetic. The modifier strictfp can be applied to classes, interfaces, and methods. A strictfp method ensures that all code in the method is executed strictly. If a class or interface is declared to be strictfp, then all code (in methods, initializers, and nested classes and interfaces) is executed strictly.
b. strictness is not inherited by the subclasses or subinterfaces.
c. Constant expressions are always evaluated strictly at compile time.
d. public strictfp class MyFPclass { // ... contents of class here ...}

25. int value = - -10; // (-(-10)) is 10
    a. blank needed to separate the unary operators; otherwise, these would be interpreted as the decrement operator --

26. Remainder Operator: %

a. the remainder can only be negative if the dividend is negative, the sign of the divisor is irrelevant.
    i. A short-cut to evaluating the remainder involving negative operands is the following: ignore the signs of the operands, calculate the remainder, and negate the remainder if the dividend is negative.
b. the remainder operator not only accepts integral operands, but floating-point operands as well.
c. float fr3 = **-7.0F % 0.0F; // NaN -7.0/0.0 is NEGATIVE_INFINITY**
d. float fr1 = 7.0F % 5.0F; // 2.0F

27. Numeric Promotions in Arithmetic Expressions
promotion of byte, short and char into int minimum
    a. byte b = 3; // int literal in range. Implicit narrowing.
    b = (byte) -b; // Explicit narrowing on assignment required.
    b. short h = 40; // OK: int converted to short. Implicit narrowing.
    h = h + 2; // **Error:** cannot assign an int to short.
    h = h + (short) 2; // Requires an additional cast. H^

28. Arithmetic Compound Assignment Operators:
x ¤= a >>>>>> x = (T) ((x) ¤ (a))
Implicit casting to the numeric T type of x

29. Implicit narrowing conversions are also applied for increment and decrement operators
30. cannot associate increment and decrement operators.
    cannot write (++(++a)). The reason is that any operand to ++ must evaluate to a variable, but the evaluation of (++a) results in a value.
31. The increment and decrement operators can also be applied to floating-point operands. In the example below, the side effect of the ++ operator is overwritten by the assignment.
    a. double x = 4.5;
       x = x + ++x; // x gets the value 10.0.
32. Care must be exercised in comparing floating-point numbers for equality, as an infinite number of floating-point values can be stored in a finite number of bits only as approximations. For example, the expression (1.0 - 2.0/3.0 == 1.0/3.0) returns false, although mathematically the result should be true.
33. int a, b, c;
    a = b = c = 5;
    boolean valid1 = **a == b == c**; // (1**) Illegal**.
    boolean valid2 = a == b && b == c; // (2) Legal.
    boolean valid3 **= a == b == true**; // (3) Legal.
34. Logical XOR | x ^ y | true if and only if one operand is true; otherwise, false.
35. In evaluation of boolean expressions involving boolean logical AND, XOR, and OR operators, both the operands are evaluated. The order of operand evaluation is always from left to right.
36. Boolean Logical Compound Assignment Operators: &=, ^=, |=
37. Short circuit Conditional Operator && (AND) and || (OR): work only on boolean operands.
    if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated. In other words, the right-hand operand is evaluated conditionally.
38. the conditional operators && and || if the evaluation of the right-hand operand is conditionally dependent on the left-hand operand. We use the logical operators & and | if both operands must be evaluated.
39. Integer Bitwise Operators: ~, &, |, ^
40. Bitwise Compound Assignment Operators: &=, ^=, |=
41. Shift Operators: <<, >>, >>>
    a. number of shifts (also called the shift distance) is given by the right-hand operand, and the value that is to be shifted is given by the left-hand operand.
    b. unary numeric promotion is applied to each operand individually.
    c. The sign bit of a byte or short value is extended to fill the higher bits when the value is promoted
    d. The value returned has the promoted type of the left-hand operand.
    e. the value of the left-hand operand is not affected by applying the shift operator.
    f. The shift distance is calculated by AND-ing the value of the right-hand operand with a mask value of 0x1f (31) if the left-hand has the promoted type int, or using a mask value of 0x3f (63) if the left-hand has the promoted type long.
    g. This effectively means masking the five lower bits of the right-hand operand in the case of an int left-hand operand, and masking the six lower bits of the right-hand operand in the case of a long left-hand operand.
    h. Thus, the shift distance is always in the range 0 to 31 when the promoted

type of left-hand operand is int, and in the range 0 to 63 when the promoted type of left-hand operand is long.

i. %32
j. the operands a and n are not affected by these three shift operators.
k. a << n: each left-shift corresponds to multiplication of the value by 2.
l. 12 << 36: 36&31: 10100&1111 = 100, i.e., 4

42. The Shift-right-with-sign-fill Operator >>:
    byte b = -42;   // 11010110
    int result = b >> 4;   // -3
43. -42 >> -4
    -4&32 : ~3: 11100&11111=11100=28
44. The Shift-right-with-zero-fill Operator >>>
    a. for positive values, the shift-right-with-zero-fill >>> and shift-right-with-sign-fill >> operators are equivalent.
45. Shift Compound Assignment Operators: <<=, >>=, >>>=

Control Flow

If-Else

The simple if statement has the following syntax:

```
if (<conditional expression>)
    <statement>
```

Since the <conditional expression> must be a boolean expression, it avoids a common programming error: using an expression of the form (a=b) as the condition, where inadvertently an assignment operator is used instead of a relational operator. The compiler will flag this as an error, unless both a and b are boolean.

switch Statement

```
switch (<non-long integral expression>) {
   case label1: <statement1>
   case label2: <statement2>
   ...
   case labeln: <statementn>
   default:  <statement>
} // end switch
```

If a and b are of type boolean, the expression (a = b) can be the conditional expression of an if statement.

The type of the switch expression is non-long integral (i.e., char, byte, short, or int).

The case labels are constant expressions whose values must be unique, meaning no duplicate values are allowed. The case label values must be assignable to the type of the switch expression (see Section 3.4, p. 48).
In particular, the case label values must be in the range of the type of the switch expression.
Note that the type of the case label cannot be boolean, long, or floating-point.

constant values should be determinable at complie time.>>>final int A=func();; will result in a compile time error >>constant expression required.

What is determinable at compile time????????/
>>>> int a=0;
final int A=a; >>>not determinable
but if u add final to >>>> final int a=0; >> this will be determinable.

```
switch(monthNumber) {
            case 12: case 1: case 2:  //>>>this is allowed
                System.out.println(" Snow in the winter.");
                break;
```

Example 5.3 Nested switch Statement

```
public class Seasons {

    public static void main(String[] args) {
        int monthNumber = 11;
        switch(monthNumber)
{                                       // (1) Outer
            case 12: case 1: case 2:
                System.out.println("Snow in the winter.");
                break;
            case 3: case 4: case 5:
                System.out.println("Green grass in spring.");
                break;
            case 6: case 7: case 8:
                System.out.println("Sunshine in the summer.");
                break;
            case 9: case 10: case 11:
         // (2)
                  switch(monthNumber) { // Nested switch
             (3) Inner
```

```
                    case 10:
                        System.out.println
("Halloween.");
                        break;
                    case 11:
                        System.out.println
("Thanksgiving.");
                        break;
                } // end nested switch
                // Always printed for case labels 9, 10,
11
                System.out.println("Yellow leaves in the
fall."); // (4)
                break;
            default:
                System.out.println(monthNumber + " is not a valid
month.");
        }
    }
}
```

Output from the program:

```
Thanksgiving.
Yellow leaves in the fall.
```

```
b3 &= b1 | b2;                  // (1) false. b3 = (b3 & (b1 | b2)).
b3 = b3 & b1 | b2;              // (2) true.  b3 = ((b3 & b1) | b2).
```

It is instructive to note how the assignments at (1) and (2) above are performed, giving different results for the same value of the operands.

The null literal can be compared, so (null != null) yields false.

The remainder operator is not limited to integral values, but can also be applied to floating-point operands.

Like the break statement, the continue statement also comes in two forms: the unlabeled and the labeled form.

```
continue;             // the unlabeled form
continue <label>;     // the labeled form
```

The continue statement can only be used in a for, while, or do-while loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible.
In the case of the while and do-while loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the <loop condition>.
In the case of the for loop, the rest of the loop body is skipped, with execution continuing with the <increment expression>.

the unlabeled form

```
class Skip {
   public static void main(String[] args) {
      for (int i = 1; i <= 5; ++i) {
         if (i == 4) continue;            // (1) Control to (2).
         // Rest of loop body skipped when i has the value 4.
         System.out.println(i + "\t" + Math.sqrt(i));
         // (2). Continue with increment expression.
      } // end for
   }
}
```

Output from the program:

```
1       1.0
2       1.4142135623730951
3       1.7320508075688772
5       2.23606797749979
```

the labeled form

```
class LabeledSkip {
   public static void main(String[] args) {
      int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
      int sum = 0;
      outer:                                          // label
      for (int i = 0; i < squareMatrix.length; ++i){          // (1)
         for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
            if (j == i) continue;                  // (3) Control to (5).
            System.out.println("Element[" + i + ", " + j + "]: " +
                               squareMatrix[i][j]);
            sum += squareMatrix[i][j];
            if (sum > 10) continue outer;          // (4) Control to (6).
            // (5) Continue with inner loop.
         } // end inner loop
         // (6) Continue with outer loop.
      } // end outer loop
      System.out.println("sum: " + sum);
   }
}
```

Output from the program:

```
Element[0, 1]: 3
Element[0, 2]: 5
Element[1, 0]: 2
Element[1, 2]: 6
Element[2, 0]: 9
```

sum: 25

break Statement

The break statement comes in two forms: the unlabeled and the labeled form.

```
break;              // the unlabeled form

break <label>;      // the labeled form
```

The unlabeled break statement terminates loops (for, while, do-while) and switch statements which contain the break statement, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, terminating the enclosing statement, with execution continuing after this statement.

block: { break block; } is a valid statement block.

```
for (int i = 1; i <= 5; ++i)
  {
    if (i == 4) break;         // (1) Terminate loop. Control to (2).
    // Rest of loop body skipped when i gets the value 4.
    System.out.println(i + "\t" + Math.sqrt(i));
} // end for
// (2) Continue here.
```

A labeled break statement can be used to terminate any labeled statement that contains the break statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

```
out:
{                       // (1) Labeled block
   // ...
   if (j == 10) break out; // (2) Terminate block. Control to (3).
   System.out.println(j);  // Rest of the block not executed if j == 10.
   // ...
}
// (3) Continue here.
```

```
class LabeledBreakOut {
   public static void main(String[] args) {
      int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
      int sum = 0;
      outer:                                          // label
      for (int i = 0; i < squareMatrix.length; ++i){          // (1)
         for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
            if (j == i) break;        // (3) Terminate this loop.
                                      //     Control to (5).
            System.out.println("Element[" + i + ", " + j + "]: " +
                               squareMatrix[i][j]);
            sum += squareMatrix[i][j];
            if (sum > 10) break outer;// (4) Terminate both loops.
                                      //     Control to (6).
         } // end inner loop
         // (5) Continue with outer loop.
      } // end outer loop
      // (6) Continue here.
      System.out.println("sum: " + sum);
   }
}
```

return Statement

The return statement is used to stop execution of a method and transfer control back to the calling code (a.k.a. the caller). The usage of the two forms of the return statement is dictated by whether it is used in a void or a non-void method. The first form does not return any value to the calling code, but the second form does. Note that the keyword void does not represent any type.

Exception:

if the code in the try block doesn't throw the specified kinda expection then there is gonna be a compile time Error.

```
class Main{
	public static void main(String arg[]){
		try{
		Cls2 obj = new Cls2(1);
		System.out.println("obj.method1:  "+obj.method1(1));
	}catch(IOException e){}
	catch(InterruptedException e){}
	catch(Exception e){}
}

exception java.lang.InterruptedException is never thrown in body of corresponding try statement
	catch(InterruptedException e){}
      ^

>>>
try{
	//code that doesn't throw any exception(checked one)
}catch(Exception e){}

//NO problem.. compile fine, as the Exception class is Super class of RUNTiMe(unchecked) 'n CHECKED Exceptions

_____________________________________
try{
	//code that doesn't throw any exception(checked one)
}catch(IOException e){}  //Catching a checked Exception, which isn't throw in try block

// Compile time error
```

The order of the catch block should be from sub-class expection to super-class exception (bottom-Up)
1. IOException
2. Exception

Reverse order will generate compile time error-
exception java.lang.InterruptedException has already been caught
	catch(InterruptedException e){}

This error will also be generated if same Exception is caught again.

```
}catch(Exception e){}
	catch(InterruptedException e){}
	catch(Exception e){}
```

---

finally will always be executed irrespective of whether an exception has occurred/not. This is also tru if a return stmt. is given in try or catch block.

```
	try{
		Cls2 obj = new Cls2(1);
		System.out.println("obj.method1:  "+obj.method1(1));
		return;
	}catch(Exception e){}
	finally{
		System.out.println("finally");
	}
```

when finally is not called:

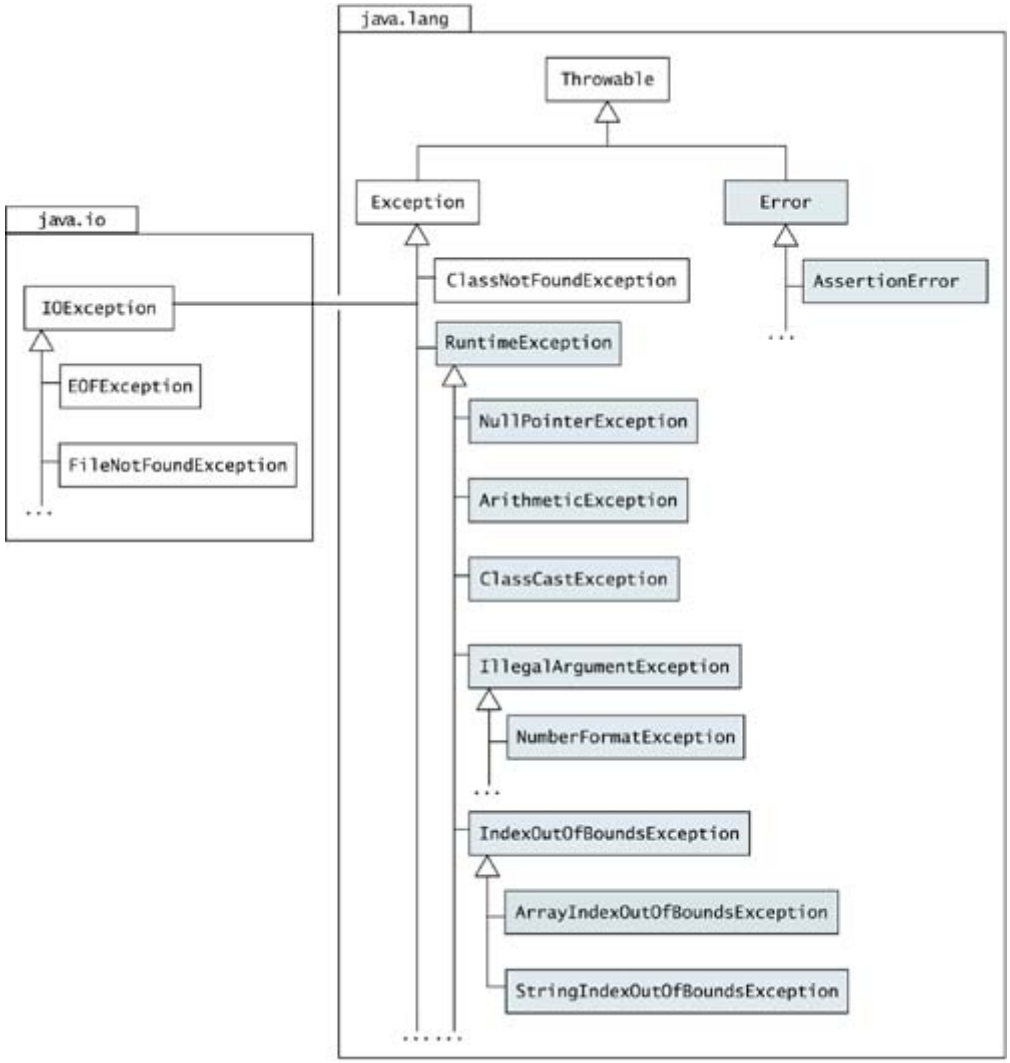
```
		try{
		Cls2 obj = new Cls2(1);
		System.out.println("obj.method1:  "+obj.method1(1));
		System.exit(0);
```

```
        }catch(Exception e){}
        finally{
                System.out.println("finally");
        }
```

5.1 Exception

- An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution.For example, a requested file cannot be found. Java provides an exception handling mechanism for systematically dealing with such error conditions.

5.2 Exception Types



Classes that are shaded (and their subclasses) represent unchecked exceptions.

The class Throwable provides the following common methods to query an exeception:

String getMessage()
Returns the detail message.

void printStackTrace()
Prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the runtime stack when the exception was thrown. The stack trace can also be written to a PrintStream or a PrintWriter by supplying such a destination as an argument to one

of the two overloaded printStackTrace() methods.

String toString()

Returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the getMessage() method.

Class Exception

The class Exception represents exceptions that a program would want to be made aware of during execution.

Class RuntimeException

Runtime exceptions like ArrayIndexOutOfBoundsException, NullPointerException, ClassCastException,IllegalArgumentException, and NumberFormatException are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class. As these runtime exceptions are usually caused by program bugs that should not occur in the first place, it is more appropriate to treat them as faults in the program design, rather than merely catching them during program execution.

Class Error

The subclass AssertionError of the java.lang.Error class is used by the Java assertion facility. These are invariably never explicitly caught and are usually irrecoverable.

Checked and Unchecked Exceptions

- Except for RuntimeException, Error, and their subclasses, all exceptions are called checked exceptions.
- The compiler ensures that if a method can throw a checked exception, directly or indirectly, then the method must explicitly deal with it.
- The method must either catch the exception and take the appropriate action, or pass the exception on to its caller
- Exceptions defined by Error and RuntimeException classes and their subclasses are known as unchecked exceptions, meaning that a method is not obliged to deal with these kinds of exceptions
- They are either irrecoverable (exemplified by the Error class) and the program should not attempt to deal with them, or they are programming errors (examplified by the RuntimeException class) and should be dealt with as such and not as exceptions.

- New exceptions usually extend the Exception class directly or one of its checked subclasses, thereby making the new exceptions checked.
- As exceptions are defined by classes, they can declare fields and methods, thus providing more information as to their cause and remedy when they are thrown and caught. The super() call can be used to set a detail message in the throwable.

```
public class EvacuateException extends Exception {
 // Data
 Date date;
 Zone zone;
 TransportMode transport;

 // Constructor
```

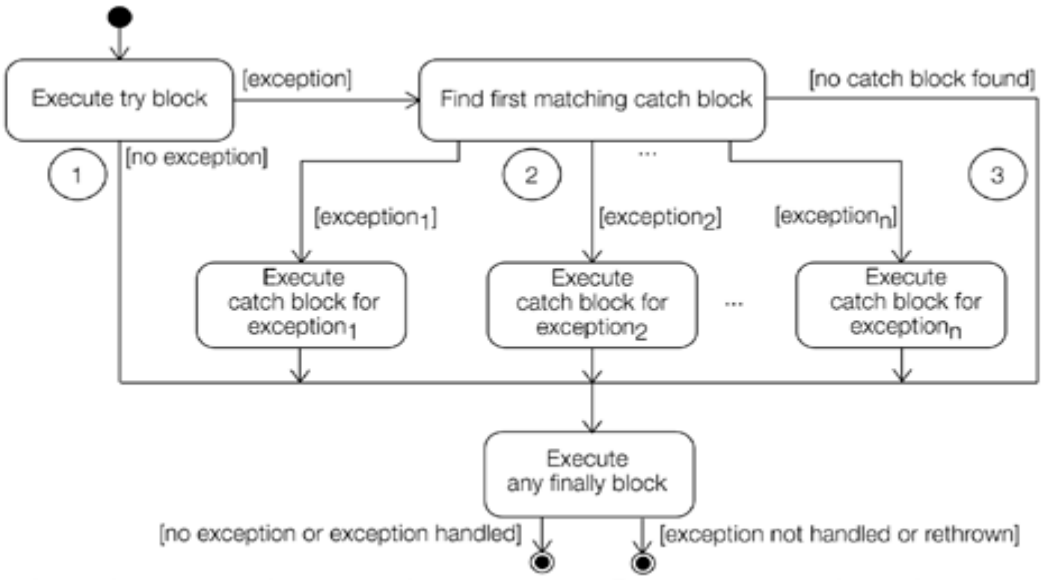
```
    public EvacuateException(Date d, Zone z, TransportMode t) {
        // Call the constructor of the superclass
        super("Evacuation of zone " + z);
        // ...
    }
    // Methods
    // ...
}
```

5.3 TRY-CATCH-FINALLY

The mechanism for handling execeptions is embedded in the try-catch-finally construct, which has the following general form:

```
try
{
        // try block
        <statements>
}
catch (<exception type-1> <parameter1>)
{
        // catch block
        <statements>
}
.
.
.
.
.
catch (<exception type-n> <parametern>)
{
        // catch block
        <statements>
}
finally
{
        // finally block
        <statements>
}
```

- Exceptions thrown during execution of the try block can be caught and handled in a catch block.
- A finally block is guaranteed to be executed, regardless of the cause of exit from the try block, or whether any catch block was executed
- The block notation is mandatory.
- For each try block there can be zero or more catch blocks, but only one finally block.
- The catch blocks and finally block must always appear in conjunction with a try block, and in the above order.
- A try block must be followed by either at least one catch block or one finally block.
- Each catch block defines an exception handler.
- The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the Throwable class or one of its subclasses.
-



- *Normal execution continues after try-catch-finally construct.* *Execution aborted and exception propagated.*

5.4 try Block

- The try block establishes a context that wants its termination to be handled. Termination occurs as a result of encountering an exception, or from successful execution of the code in the try block.
- For all exits from the try block, except those due to exceptions, the catch blocks are skipped and control is transferred to the finally block, if one is specified.
- For all exits from the try block resulting from exceptions, control is transferred to the catch blocks—if any such blocks are specified—to find a matching catch block. If no catch block matches the thrown exception, control is transferred to the finally block, if one is specified.

5.5 catch Block

- Only an exit from a try block resulting from an exception can transfer control to a catch block. A catch block can only catch the thrown exception if the exception is assignable to the parameter in the catch block. The code of the first such catch block is executed and all other catch blocks are ignored.
- On exit from a catch block, normal execution continues unless there is any pending exception that has been thrown and not handled. If this is the case, the method is aborted and the exception is propagated up the runtime stack as explained earlier.
- After a catch block has been executed, control is always transferred to the finally block, if one is specified. This is always true as long as there is a finally block, regardless of whether the catch block itself throws an exception.

EXAMPLE:

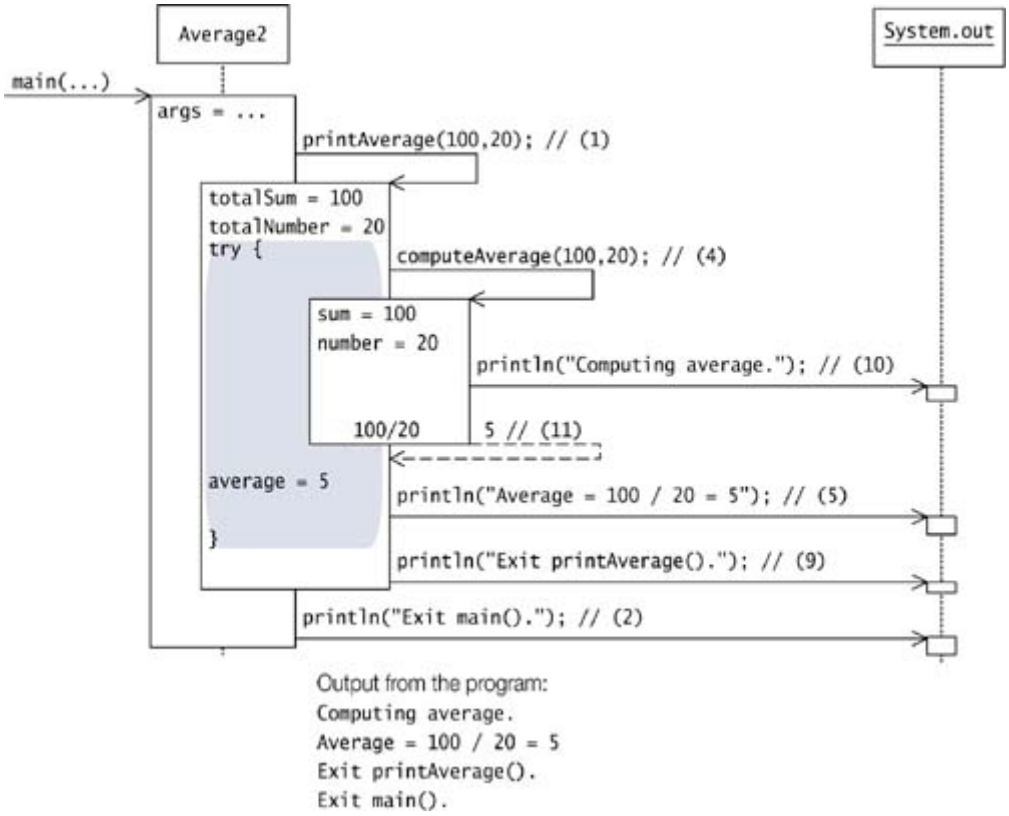
```
public class Average3 {

   public static void main(String[] args) {
      try {                                    // (1)
         printAverage(100, 0);                 // (2)
      } catch (ArithmeticException ae) {       // (3)
         ae.printStackTrace();                 // (4)
         System.out.println("Exception handled in " +
                            "main().");        // (5)
```

```
        }
        System.out.println("Exit main().");                  // (6)
    }

    public static void printAverage(int totalSum, int totalNumber) {
        try {                                               // (7)
            int average = computeAverage(totalSum, totalNumber);  // (8)
            System.out.println("Average = " +                // (9)
                totalSum + " / " + totalNumber + " = " + average);
        } catch (IllegalArgumentException iae) {            // (10)
            iae.printStackTrace();                          // (11)
            System.out.println("Exception handled in " +
                               "printAverage().");          // (12)
        }
        System.out.println("Exit printAverage().");         // (13)
    }

    public static int computeAverage(int sum, int number) {
        System.out.println("Computing average.");           // (14)
        return sum/number;                                  // (15)
    }
}
```

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:30)
        at Average3.printAverage(Average3.java:17)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

•



Output from the program:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

- The scope of the argument name in the catch block is the block itself.
- The javac compiler also complains if a catch block for a superclass exception shadows the catch block for a subclass exception, as the catch block of the subclass exception will never be executed.

```
// Compiler complains
catch (Exception e)
{
        // (1) superclass
        System.out.println(e);
}
catch (ArithmeticException e)
{
        // (2) subclass
        System.out.println(e);
}
...
```

- The catch block can only catch those exception which might be thrown in the body of try block. Otherwise compile time error - Unreachable catch block for that **Exception**. This exception is never thrown from the try statement.

```
try
{

}
catch(IOException ioe)    //        Error-  Unreachable catch block for  IOException. This exception is never thrown from the try statement
{
}
```

5.6 FINALLY BLOCK

- If the try block is executed, then the finally block is guaranteed to be executed, regardless of whether any catch block was executed. Since the finally block is always executed before

control transfers to its final destination, it can be used to specify any clean-up code (for example, to free resources such as, files, net connections).

- If the finally block neither throws an exception nor executes a control transfer statement like a return or a labeled break, then the execution of the try block or any catch block determines how execution proceeds after the finally block.

  - If there is no exception thrown during execution of the try block or the exception has been handled in a catch block, then normal execution continues after the finally block.

  - If there is any pending exception that has been thrown and not handled (either due to the fact that no catch block was found or the catch block threw an exception), the method is aborted and the exception is propagated after the execution of the finally block.

- If the finally block throws an exception, then this exception is propagated with all its ramifications—regardless of how the try block or any catch block were executed. In particular, the new exception overrules any previously unhandled exception.

- If the finally block executes a control transfer statement such as, a return or a labeled break, then this control transfer statement determines how the execution will proceed—regardless of how the try block or any catch block were executed. In particular, a value returned by a return statement in the finally block will supersede any value returned by a return statement in the try block or a catch block.
- Finally block will not be executed only when **System.exit(6);** is used explicitly.
- Finally block does not handle exception thrown in the try block.It has to be handled by catch block only or pass it to the calling method by using **throws**.

Example 5.13 try-finally Construct

```
public class Average5 {

   public static void main(String[] args) {
      printAverage(100, 0);                          // (1)
      System.out.println("Exit main().");                 // (2)
   }

   public static void printAverage(int totalSum, int totalNumber) {
      try {                                          // (3)
         int average = computeAverage(totalSum, totalNumber);  // (4)
         System.out.println("Average = " +                  // (5)
            totalSum + " / " + totalNumber + " = " + average);
      } finally {                                    // (6)
         System.out.println("Finally done.");
      }
      System.out.println("Exit printAverage().");          // (7)
   }

   public static int computeAverage(int sum, int number) {
      System.out.println("Computing average.");            // (8)
      return sum/number;                             // (9)
   }
}
```

Output from the program:

```
Computing average.
Finally done.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average5.computeAverage(Average5.java:21)
        at Average5.printAverage(Average5.java:10)
        at Average5.main(Average5.java:4)
```

- the execution of a control transfer statement such as a return in the finally block affects the program execution.The first output from the program shows that the average is computed, but the value returned is from the return statement at (8) in the finally block, not from the return statement at (6) in the try block. The second output shows that the ArithmeticException thrown in the computeAverage() method and propagated to the printAverage() method, is nullified by the return statement in the finally block. Normal execution continues after the return statement at (8), with the value 0 being returned from the printAverage() method.

```
public class Average6 {

   public static void main(String[] args) {
      System.out.println("Average: " + printAverage(100, 20));  // (1)
      System.out.println("Exit main().");                       // (2)
   }

   public static int printAverage(int totalSum, int totalNumber) {
      int average = 0;
      try {                                              // (3)
         average = computeAverage(totalSum, totalNumber);      // (4)
         System.out.println("Average = " +                     // (5)
            totalSum + " / " + totalNumber + " = " + average);
         return average;                                 // (6)
      } finally {                                        // (7)
         System.out.println("Finally done.");
         return average*2;                               // (8)
      }
   }

   public static int computeAverage(int sum, int number) {
      System.out.println("Computing average.");               // (9)
      return sum/number;                                 // (10)
   }
}
```

Output from the program, with call printAverage(100, 0) in (1):

```
Computing average.
Finally done.
Average: 0
Exit main().
```

5.7 throw Statement

A program can explicitly throw an exception using the throw statement. The general format of the throw statement is as follows:

throw <object reference expression>; **throw new NullPointerException**

- The compiler ensures that the <object reference expression> is of type Throwable class or one of its subclasses.At runtime a NullPointerException is thrown if the <object reference expression> is null. This ensure that a Throwable will always be propagated. A detail message is often passed to the constructor when the exception object is created.

```
throw new ArithmeticException("Integer division by 0");
```

  - Ø When an exception is thrown, normal execution is suspended.
  - Ø The runtime system proceeds to find a catch block that can handle the exception.
  - Ø The search starts in the context of the current try block, propagating to any enclosing try blocks and through the runtime stack to find a handler for the exception.
  - Ø Any associated finally block of a try block encountered along the search path is executed.
  - Ø If no handler is found, then the exception is dealt with by the default exception handler at the top level.
  - Ø If a handler is found, execution resumes with the code in its catch block.

```
o  Immediately after throw keyword, there can't be any code in that block.
o      try
o      {
o              System.out.println("In try");
o              throw new IOException();
o              if(true)                        // compile time error- unreachable code
o              {
o                      System.out.println("In try");
o              }
o              System.out.println("In try");
o              return 0;
o      }
o      catch(IOException e)
o      {
o              System.out.println("In catch");
o              return 1;
o      }
```

Example 5.15 Throwing Exceptions

```
public class Average7 {

   public static void main(String[] args) {
      try {                                    // (1)
         printAverage(100, 0);                 // (2)
      } catch (ArithmeticException ae) {       // (3)
         ae.printStackTrace();                 // (4)
         System.out.println("Exception handled in " +     // (5)
                            "main().");
      } finally {
         System.out.println("Finally in main().");        // (6)
      }
      System.out.println("Exit main().");      // (7)
   }

   public static void printAverage(int totalSum, int totalNumber) {
```

```
        try {                                           // (8)
            int average = computeAverage(totalSum, totalNumber);   // (9)
            System.out.println("Average = " +                    // (10)
                totalSum + " / " + totalNumber + " = " + average);
        } catch (IllegalArgumentException iae) {                 // (11)
            iae.printStackTrace();                               // (12)
            System.out.println("Exception handled in " +          // (13)
                               "printAverage().");
        } finally {
            System.out.println("Finally in printAverage().");     // (14)
        }
        System.out.println("Exit printAverage().");              // (15)
    }

    public static int computeAverage(int sum, int number) {
        System.out.println("Computing average.");
        if (number == 0)                                         // (16)
            throw new ArithmeticException("Integer division by 0");// (17)
        return sum/number;                                       // (18)
    }
}
```

Output from the program:

```
Computing average.
Finally in printAverage().
java.lang.ArithmeticException: Integer division by 0
        at Average7.computeAverage(Average7.java:35)
        at Average7.printAverage(Average7.java:19)
        at Average7.main(Average7.java:6)
Exception handled in main().
Finally in main().
Exit main().
```

5.8 throws Clause

- A throws clause can be specified in the method protoype.

... someMethod(...)
  throws <ExceptionType1>, <ExceptionType2>,..., <ExceptionTypen> { ... }

  - Each <ExceptionTypei> declares a checked exception.
  - The compiler enforces that the checked exceptions thrown by a method are limited to those specified in its throws clause.
  - Of course, the method can throw exceptions that are subclasses of the checked exceptions in the throws clause. This is permissable since exceptions are objects, and a subclass object can polymorphically act as an object of its superclass
  - The throws clause can have unchecked exceptions specified, but this is seldom used and the compiler does not check them.

- Any method that can cause a checked exception to be thrown, either directly by using the throw statement or indirectly by invoking other methods that can throw such an exception,

must deal with the exception in one of three ways. It can

- use a try block and catch the exception in a handler and deal with it
- use a try block and catch the exception in a handler, but throw another exception that is either unchecked or declared in its throws clause
- explicitly allow propagation of the exception to its caller by declaring it in the throws clause of its method protoype

Native methods can also declare checked exceptions in their throws clause, but the compiler is not able to check them for consistency.

The exception type specified in the throws clause in the method protoype can be a superclass type of the actual exceptions thrown, that is, the exceptions thrown must be assignable to the type of the exceptions specified in the throws clause. If a method can throw exceptions of the type A, B, and C where these are subclasses of type D, then the throws clause can either specify A, B, and C or just specify D. In the printAverage() method, the method protoype could specify the superclass Exception of the subclass IntegerDivisionByZero in a throws clause.

```
public static void printAverage(int totalSum, int totalNumber)throws Exception
{ /* ... */ }
```

A subclass can override a method defined in its superclass by providing a new implementation (see Section 6.2, p. 233). What happens when an inherited method with a list of exceptions in its throws clause is overridden in a subclass? The method definition in the subclass can only specify a subset of the checked exception classes (including their subclasses) from the throws clause of the inherited method in the superclass. This means that an overriding method cannot allow more checked exceptions in its throws clause than the inherited method does. Allowing more checked exceptions in the overriding method would create problems for clients who already deal with the exceptions specified in the inherited method. Such clients would be ill prepared if an object of the subclass (under the guise of polymorphism) threw a checked exception they were not prepared for.

```
class A
{
   // ...
   protected void superclassMethodX()
      throws FirstException, SecondException, ThirdException {/* ... */} // (1)
   // ...
}

class B extends A
{
   // ...
   protected void superclassMethodX()
      throws FirstException, ThirdException { /* ... */ }              // (2)
   // ...

}
```

In the previous code, the method superclassMethodX in superclass A is overridden in subclass B. The throws clause of the method in subclass B at (2) is a subset of the exceptions specified for the

method in the superclass at (1).

The LAW: Handle or **Declare**.
Handle (using catch) or **Declare** (using throws).A try without a catch doesn't satisfy the **Handle or Declare law.** A try with only a finally (no catch) must still declare the exception.

Exception Rules:

1. There cant be catch or finally without try.
2. We cannot put code between try and catch.
3. A try must be followed by either at least one catch or a finally.
4. A try with only a finally (no catch) must still declare the exception.

```
package fundamental;

import java.io.IOException;

public class AQuestion
{
	public static void main(String args[])
	{
		System.out.println("Before Try");
		try
		{
			System.out.println("In Try");
		}
		catch (IOException fnfe)
		{
		}
```

Unreachable catch block for IOException. This exception is never thrown from the try statement body

```
		catch (Exception fnfe)
		{
		}
		catch (Throwable t)
		{
			System.out.println("Inside Catch");
		}
		System.out.println("At the End");

	}
}
```

```
import java.io.IOException;
public class Question72
{
public Question72() throws IOException
{
throw new IOException();
}
}
```

Assume that the defination of Question72E begins with the line

`public class Question72E extends Question72`

It is required that none of the constructors of Question72E should throw any checked exception.

It cannot be done in the Java Laungage with the above definition of the base class

Assertion is the **expected state** at a particular point in the program.
:
assert c<0 : "c is greater than 0"; // If the c>=0 a new object of AssertionError is thrown<-- new AssertionError("msg");
: // these stmts will be executed if the condition is true
:

java.lang.AssertionError --->java.lang.Error

java.lang.AssertionError :
Has 7 parameterized Constructors. -- Single parameter Consturctors <<< 6 for primitive Datatypes +1 Object reference[Not String]
1 Default Constructor.

Objective of Assertion is: Debugging << so are rarely caught

Can be enabled/disabled at the compile time. -ea -da [System Classes]
for System Classes: -esa -dsa

Exception

1. An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution.
2. To throw an exception is to signal that an unexpected error condition has occurred.
3. To catch an exception is to take appropriate action to deal with the exception.
4. An exception is caught by an exception handler, and the exception need not be caught in the same context that it was thrown in.
5. Exceptions in Java are objects.
6. All exceptions are derived from the `java.lang. Throwable` class
7. The two main subclasses **`Exception` and `Error`** constitute the main categories of throwables.
8. Except for **`RuntimeException`**, **`Error`**, and their subclasses, all exceptions are called checked exceptions.
9. Exceptions defined by **`Error` and `RuntimeException`** classes and their subclasses are known as **unchecked exceptions.**
10. **New exceptions** are usually defined to provide fine-grained categorization of exceptional conditions, instead of using existing exception classes with descriptive detail messages to differentiate between the conditions.
11. **New exceptions usually extend the `Exception` class directly or one of its checked subclasses**, thereby making the new exceptions checked.
12. **New exception class** Can declare fields and methods, thus providing more information as to their cause and remedy when they are thrown and caught.
13. The `super()` call can be used to set a detail message in the throwable
14. The mechanism for handling execeptions is embedded in the `try-catch-finally` construct, which has the following general form:
15. try { // try block

```
 <statements>
   } catch (<exception type1> <parameter1>) {  // catch block
      <statements>
   }
   ...
     catch (<exception typen> <parametern>) {  // catch block
       <statements>
   } finally {                                  // finally block
      <statements>
   }
```

16. A **`finally`** block is **guaranteed to be executed**, regardless of the cause of exit from the `try` block, or whether any `catch` block was executed.
17. **The block notation is mandatory.**
18. For **each `try` block there can be zero or more `catch` blocks, but only one `finally` block.**
19. **The `catch` blocks and `finally` block must always appear in conjunction with a `try` block, and in the above order.**
20. **A `try` block must be followed by either at least one `catch` block or one `finally` block.**
21. Each `catch` block defines an exception handler.
22. The header of the `catch` block takes exactly one argument, which is the exception its block is willing to handle.
23. The exception must be of the `Throwable` class or one of its subclasses.
24. // **Compiler complains**

```
        catch (Exception e) {                   // (1) superclass
            System.out.println(e);
        } catch (ArithmeticException e) {       // (2) subclass
            System.out.println(e);
        }
```

```
...
```

25. A `try-finally` construct can be used to control the interplay between two actions that must be executed in the right order, possibly with other intervening actions.
26. **Example**

```
int sum = -1;
try {
    sum = sumNumbers();
    // other actions
} finally {
    if (sum >= 0) calculateAverage();
}
```

a. The code above guarantees that if the `try` block is entered `sumNumbers()` will be executed first and then later `calculateAverage()` will be executed in the `finally` block, regardless of how execution proceeds in the `try` block. As the operation in `calculateAverage()` is dependent on the success of `sumNumbers()`, this is checked by the value of the `sum` variable before calling `calculateAverage()`. `catch` blocks can, of course, be included to handle any exceptions.

27. If the `finally` block neither throws an exception nor executes a control transfer statement like a `return` or a labeled `break`, then the execution of the `try` block or any `catch` block determines how execution proceeds after the `finally` block .

b. If there is no exception thrown during execution of the `try` block or the exception has been handled in a `catch` block, then normal execution continues after the `finally` block.

c. If there is any pending exception that has been thrown and not handled (either due to the fact that no `catch` block was found or the `catch` block threw an exception), the method is aborted and the exception is propagated after the execution of the `finally` block.

28. If the `finally` block throws an exception, then this exception is propagated with all its ramifications—regardless of how the `try` block or any `catch` block were executed.

29. In particular, the new exception overrules any previously unhandled exception.

30. **If the `finally` block executes a control transfer statement such as, a `return` or a labeled `break`, then this control transfer statement determines how the execution will proceed—regardless of how the `try` block or any `catch` block were executed.**
31. **value returned by a `return` statement in the `finally` block will supersede any value returned by a `return` statement in the `try` block or a `catch` block**.

32. **`throws`** clause can be specified in the method protoype.

```
... someMethod(...)
  throws <ExceptionType1>, <ExceptionType2>, ..., <ExceptionTypen> { ... }
```

Each <ExceptionType$_1$> declares a checked exception.

33. any method that can cause a checked exception to be thrown, either directly by using the `throw` statement or indirectly by invoking other methods that can throw such an exception, must deal with the exception in one of three ways.
34. It can

d. use a `try` block and catch the exception in a handler and deal with it

e. use a `try` block and catch the exception in a handler, but throw another exception that is either unchecked or declared in its `throws` clause

f. explicitly allow propagation of the exception to its caller by declaring it in the `throws` clause of its method prototype

35. program can explicitly throw an exception using the **`throw` statement**.

36. `throw` <object reference expression>`;`

37. <object reference expression> should be of type `Throwable` class or one of its subclasses.

38. At runtime a `NullPointerException` is thrown if the <object reference expression> is `null`.

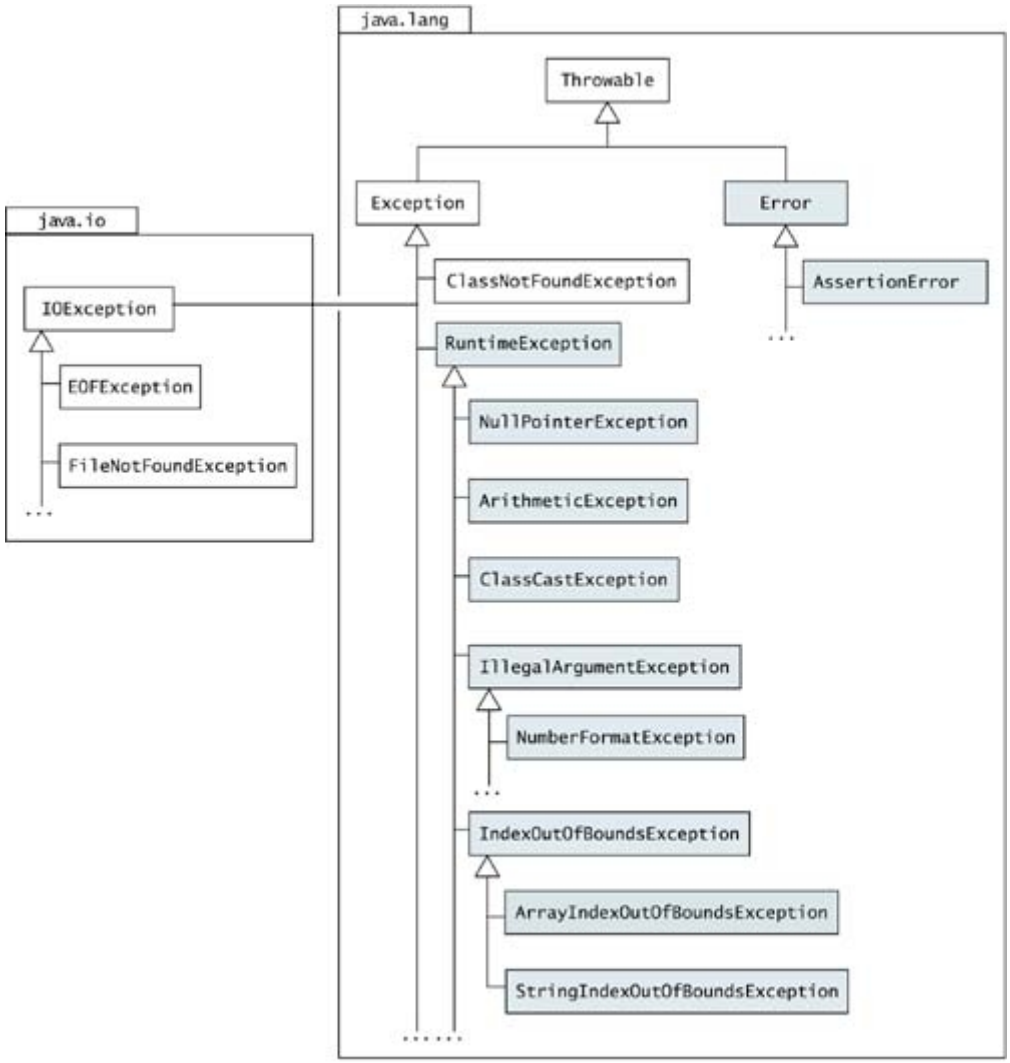
39. This ensure that a `Throwable` will always be propagated. A detail message is often passed to the constructor when the exception object is created.

g. throw new ArithmeticException("Integer division by 0");

40. When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a `catch` block that can handle the exception.

41. The search starts in the context of the current `try` block, propagating to any enclosing `try` blocks and through the runtime stack to find a handler for the exception.

42. Any associated `finally` block of a `try` block encountered along the search path is executed



Classes that are shaded (and their subclasses) represent unchecked exceptions.

The class Throwable provides the following common methods to query an exeception:

String getMessage()
Returns the detail message.

void printStackTrace()
Prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the runtime stack when the exception was thrown. The stack trace can also be written to a PrintStream or a PrintWriter by supplying such a destination as an argument to one of the two overloaded printStackTrace() methods.

String toString()

Returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the getMessage() method.

Class /Method Declaration

1. A class declaration introduces a new reference type.

   a. Syntax>>.class

   i. **<class modifiers>** `class` <class name>
   <extends clause> <implements clause> `// Class header`
   `{ // Class body`
   <field declarations>
   <method declarations>
   <nested class declarations>
   <nested interface declarations>
   <constructor declarations>
   <initializer blocks>
   `}`

   class header can specify the following information:

   ii. scope or accessibility modifier

   iii. additional class modifiers

   iv. any class it extends

   v. any interfaces it implements

   b. Syntax>> method

   i. <**method modifiers**> <**return type**> <method name> `(`< **formal parameter list**>`)`
   <throws clause> `// Method prototype`
   `{ // Method body`
   <local variable declarations>
   <nested local class declarations>
   <statements>
   `}`

   ii. In addition to the name of the method, the method prototype can specify the following information:

   iii. scope or accessibility modifier

   iv. additional method modifiers

   v. type of the return value, or `void` if the method does not return any value

   vi. formal parameter list

   vii. checked exceptions thrown by the method in a `throws` clause

   viii. The signature of a method comprises the method name and the formal parameter list only.

2. Class names and method names exist in different namespaces. Thus, there are no name conflicts

3. Modifiers for classes

    c. Public

    d. Default/friendly

    e. Abstract

    f. Final

4. Abstract class cannot be both final and abstract

5. Interfaces, which are inherently `abstract`, thus cannot be declared `final`

6. Statements can be grouped into various categories.

    g. Variable declarations with explicit initialization of the variables are called **declaration statements .**

    h. Control flow statements

    i. expression statements.

7. An expression statement is an expression terminated by a semicolon.

8. The expression is evaluated for its side effect and its value discarded.

9. Only certain types of expressions have meaning as statements.

10. They include the following:

    j. Assignments

    k. Increment and decrement operators

    l. Method calls

    m. Object creation expression with the `new` operator

11. A solitary semicolon denotes the empty statement that does nothing.

12. A block, `{}`, is a **compound statement** which can be used to group zero or more local declarations and statements (see Section 4.5, p. 123).

13. Blocks can be nested, since a block is a statement that can contain other statements.

14. A block can be used in any context where a simple statement is permitted.

15. **`Modifiers for methods`**

`public`	Accessible everywhere.
`protected`	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.

default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
`private`	Only accessible in its own class and not anywhere else.

16.

Table 4.5. Summary of Other Modifiers for Members

Modifiers	Fields	Methods
`static`	Defines a class variable.	Defines a class method.
`final`	Defines a constant.	The method cannot be overridden.
`abstract`	N/A	No method body is defined. Its class must also be designated `abstract`.
`synchronized`	N/A	Only one thread at a time can execute the method.
`native`	N/A	Declares that the method is implemented in another language.
`transient`	The value in the field will not be included when the object is serialized.	Not applicable.
`volatile`	The compiler will not attempt to optimize access to the value in the field.	Not applicable.