Object Lifetime

Garbage Collection

- Storage for objects is allocated in a designated part of the memory called the heap which has a finite size.
- Garbage collection is a process of managing the heap efficiently; i.e., reclaiming memory occupied by objects that are no longer needed and making it available for new objects.

Garbage Collection

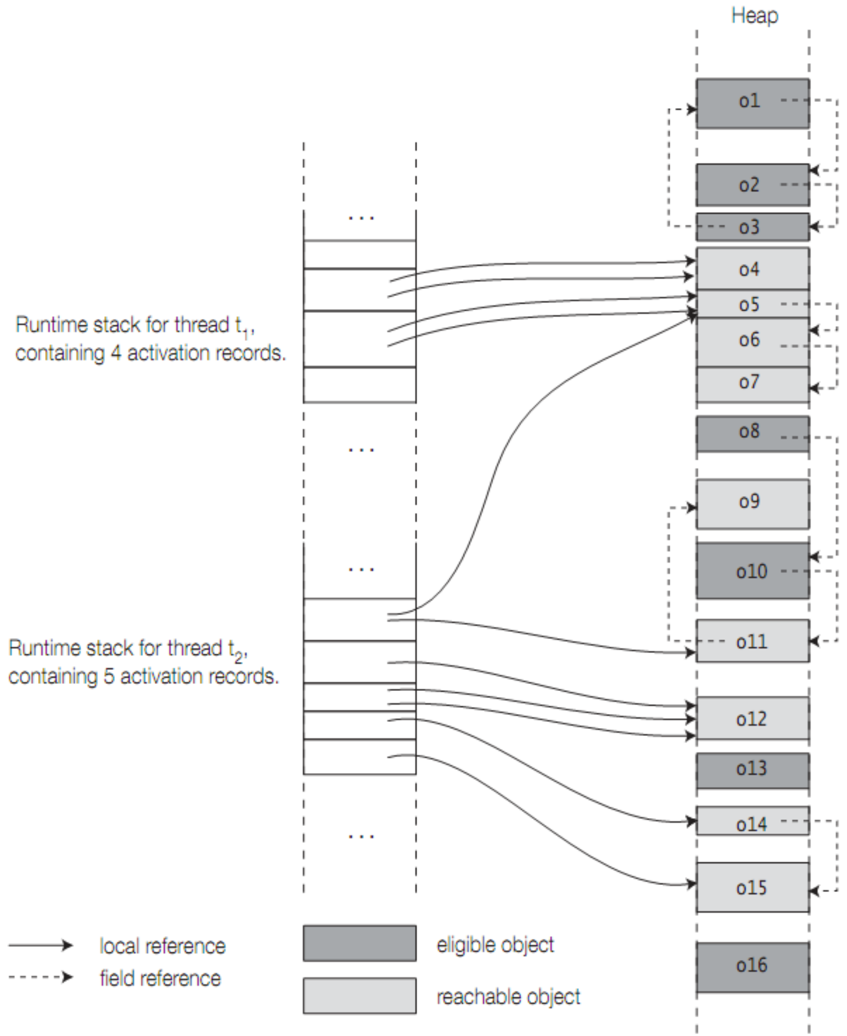
- Objects allocated on the heap (through the new operator) are administered by the automatic garbage collector.
- The automatic garbage collection scheme guarantees that a reference to an object is always valid while the object is needed by the program.
- The object will not be reclaimed, leaving the reference dangling.

Garbage Collection

- By relying on the automatic garbage collector, a Java program also forfeits any significant influence on the garbage collection of its objects.
- However, this price is insignificant when compared to the cost of putting the code for object management in place and plugging all the memory leaks.
- Time-critical applications should bear in mind that the automatic garbage collector runs as a background task and may have a negative impact on their performance.

Reachable Objects

- An automatic garbage collector essentially performs two tasks:
  - ➢ decides if and when memory needs to be reclaimed
  - ➢ finds objects that are no longer needed by the program and reclaims their storage
- 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


Heap
o1
o2
o3
o4
o5
o6
o7
o8
o9
o10
o11
o12
o13
o14
o15
o16
...
Runtime stack for thread t_1, containing 4 activation records.
Runtime stack for thread t_2, containing 5 activation records.
local reference
field reference
eligible object
reachable object

Two live threads (t1 and t2) and their respective runtime stacks with the activation records. The diagram shows which objects in the heap are referenced by local references in the method activation records. The diagram also shows field references in objects, which refer to other objects in the heap.

Some objects have several aliases.

- Objects o4, o5, o11, o12, o14, and o15 all have reachable references.
- Objects o13 and o16 have no reachable references and are, therefore, eligible for garbage collection.

Reachable Objects

- Although the objects o1, o2, and o3 form a circular list, they do not have any reachable references. Thus, these objects are all eligible for garbage collection.
- On the other hand, the objects o5, o6, and o7 form a linear list, but they are all reachable, as the first object in the list, o5, is reachable.
- The objects o8, o10, o11, and o9 also form a linear list (in that order), but not all objects in the list are reachable. Only the objects o9 and o11 are reachable, as object o11 has a reachable reference.
- The objects o8 and o10 are eligible for garbage collection.

Lifetime of an object

- The lifetime of an object is the time from its creation to the time it is garbage collected. Under normal circumstances, an object is accessible from the time when it is created to the time when it is unreachable.
- The lifetime of an object can also include a period when it is eligible for garbage collection, waiting for its storage to be reclaimed.
- The finalization mechanism in Java does provide a means for resurrecting an object after it is eligible for garbage collection, but the finalization mechanism is rarely used for this purpose.

Facilitating Garbage Collection

- The automatic garbage collector determines which objects are not reachable and, therefore, eligible for garbage collection. It will certainly go to work if there is an imminent memory shortage.
- Automatic garbage collection should not be perceived as a license for uninhibited creation of objects and then forgetting about them.

Facilitating Garbage Collection

- Certain objects, such as files and network connections, can tie up resources and should be disposed of properly when they are no longer needed.
- In most cases, the finally block in the **try-catch-finally** construct provides a convenient facility for such purposes, as it will always be executed, thereby ensuring proper disposal of any unwanted resources.

Facilitating Garbage Collection

- To optimize its memory footprint, a live thread should only retain access to an object as long as the object is needed for its execution.
- The program can allow objects to become eligible for garbage collection as early as possible by removing all references to the object when it is no longer needed.

Facilitating Garbage Collection

- 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 out of the method.
- This can occur if a reference value is returned from the method, passed as argument to another method that records the reference value, or thrown as an exception.
- However, a method need not always leave objects to be garbage collected after its termination. It can facilitate garbage collection by taking suitable action, for example, by nulling references.

```
import java.io.*;
class WellBehavedClass {
  // ...
  void wellBehavedMethod() {
    File aFile;
    long[] bigArray = new long[20000];
    // ... uses local variables ...
    // Does cleanup (before starting something extensive)
    aFile = null;                  // (1)
    bigArray = null;               // (2)
    // Start some other extensive activity
    // ...
  }
  // ...
}
```

- In the previous code, the local variables are set to null after use at (1) and (2), before starting some other extensive activity. This makes the objects denoted by the local variables eligible for garbage collection from this point onward, rather than after the method terminates.
- This optimization technique of nulling references need only be used as a last resort when resources are scarce.

Some other techniques to facilitate garbage collection

- When a method returns a reference value and the object denoted by the value is not needed, not assigning this value to a reference also facilitates garbage collection.
- If a reference is assigned a new value, the object that was previously denoted by the reference prior to the assignment can become eligible for garbage collection.
- Removing reachable references to a composite object can make the constituent objects become eligible for garbage collection, as explained earlier

Example: Garbage Collection Eligibility

```
class HeavyItem {                                 // (1)
  int[]     itemBody;
  String    itemID;
  HeavyItem nextItem;
  HeavyItem(String ID, HeavyItem itemRef) {      // (2)
    itemBody = new int[100000];
    itemID   = ID;
    nextItem = itemRef;
  }
  protected void finalize() throws Throwable { // (3)
    System.out.println(itemID + ": recycled.");
    super.finalize();
  }
}
```

```java
public class RecyclingBin {
  public static HeavyItem createHeavyItem(String itemID) {          // (4)
    HeavyItem itemA = new HeavyItem(itemID + " local item", null); // (5)
    itemA = new HeavyItem(itemID, null);                           // (6)
    System.out.println("Return from creating HeavyItem " + itemID);
    return itemA;                                                  // (7)
  }
  public static HeavyItem createList(String listID) {              // (8)
    HeavyItem item3 = new HeavyItem(listID + ": item3", null);     // (9)
    HeavyItem item2 = new HeavyItem(listID + ": item2", item3);    // (10)
    HeavyItem item1 = new HeavyItem(listID + ": item1", item2);    // (11)
    System.out.println("Return from creating list " + listID);
    return item1;                                                  // (12)
  }
  public static void main(String[] args) {                         // (13)
    HeavyItem list = createList("X");                              // (14)
    list = createList("Y");                                        // (15)
    HeavyItem itemOne = createHeavyItem("One");                    // (16)
    HeavyItem itemTwo = createHeavyItem("Two");                    // (17)
    itemOne = null;                                                // (18)
    createHeavyItem("Three");                                      // (19)
    createHeavyItem("Four");                                       // (20)
    System.out.println("Return from main().");
  }
}
```

Possible output from the program:

```
Return from creating list X
Return from creating list Y
X: item3: recycled.
X: item2: recycled.
X: item1: recycled.
Return from creating HeavyItem One
Return from creating HeavyItem Two
Return from creating HeavyItem Three
Three local item: recycled.
Three: recycled.
Two local item: recycled.
Return from creating HeavyItem Four
One local item: recycled.
One: recycled.
Return from main().
```

Object Finalization

- Object finalization provides an object with a last resort to undertake any action before its storage is reclaimed.
- The automatic garbage collector calls the finalize() method in an object that is eligible for garbage collection before actually destroying the object.
- The finalize() method is defined in the Object class.

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

Object Finalization

- An implementation of the finalize() method is called a *finalizer*.
- A subclass can override the *finalizer* from the Object class in order to take more specific and appropriate action before an object of the subclass is destroyed.
- Note that the overridden method cannot narrow the visibility of the method and it must be declared either **protected** or **public**.

Object Finalization

- A finalizer can, like any other method, catch and throw exceptions. However, any exception thrown but not caught by a finalizer that is called by the garbage collector is ignored, and the finalization of this object is terminated.
- The finalizer is only called once on an object, regardless of whether any exception is thrown during its execution. In case of finalization failure, the object still remains eligible for disposal at the discretion of the garbage collector (unless it has been resurrected, as explained in the next subsection).
- Since there is no guarantee that the garbage collector will ever run, there is also no guarantee that the finalizer will ever be called.

Object Finalization

In the following code, the finalizer at (1) will take appropriate action if and when called on objects of the class before they are garbage collected, ensuring that the resource is freed. Since it is not guaranteed that the finalizer will ever be called at all, a program should not rely on the finalization to do any critical operations.

```java
public class AnotherWellBehavedClass {
  SomeResource objRef;
  // ...
  protected void finalize() throws Throwable {         // (1)
    try {                                              // (2)
      if (objRef != null) objRef.close();
    } finally {                                        // (3)
      super.finalize();                                // (4)
    }
  }
}
```

Object Finalization

- Note that an enum type cannot declare a finalizer.
- Therefore, an enum constant may never be finalized.

Finalizer Chaining

- Unlike subclass constructors, overridden finalizers are not implicitly chained. Therefore, the finalizer in a subclass should explicitly call the finalizer in its superclass as its last action, as shown at (4) in the previous code.
- The call to the finalizer of the superclass is in a finally block at (3), guaranteed to be executed regardless of any exceptions thrown by the code in the try block at (2).

Example: Using Finalizers

```
class BasicBlob {                                         // (1)
  static    int idCounter;
  static    int population;
  protected int blobId;
  BasicBlob() {
    blobId = idCounter++;
    ++population;
  }
  protected void finalize() throws Throwable {         // (2)
    --population;
    super.finalize();
  }
}
```

```
class Blob extends BasicBlob {                              // (3)
  int[] size;
  Blob(int bloatedness) {                                   // (4)
    size = new int[bloatedness];
    System.out.println(blobId + ": Hello");
  }
  protected void finalize() throws Throwable {               // (5)
    System.out.println(blobId + ": Bye");
    super.finalize();
  }
}
public class Finalizers {
  public static void main(String[] args) {                   // (6)
    int blobsRequired, blobSize;
    try {
      blobsRequired = Integer.parseInt(args[0]);
      blobSize   = Integer.parseInt(args[1]);
    } catch(IndexOutOfBoundsException e) {
      System.err.println("Usage: Finalizers <num of blobs> <blob size>");
      return;
    }
    for (int i=0; i<blobsRequired; ++i) {                       // (7)
      new Blob(blobSize);
    }
    System.out.println(BasicBlob.population + " blobs alive"); // (8)
  }
}
```

Example: Using Finalizers

Running the program with the command

```
>java Finalizers 5 500000
```

resulted in the following output:

```
0: Hello
1: Hello
2: Hello
0: Bye
1: Bye
2: Bye
3: Hello
4: Hello
2 blobs alive
```

Invoking Garbage Collection Programmatically

- Although Java provides facilities to invoke the garbage collection explicitly, there are no guarantees that it will be run.
- The program can only request that garbage collection be performed, but there is no way that garbage collection can be forced.
- The System.gc() method can be used to request garbage collection, and the System.runFinalization() method can be called to suggest that any pending finalizers be run for objects eligible for garbage collection.
- Alternatively, corresponding methods in the Runtime class can be used. A Java application has a unique Runtime object that can be used by the application to interact with the JVM. An application can obtain this object by calling the method Runtime.getRuntime(). The Runtime class provides various methods related to memory issues.

Invoking Garbage Collection Programmatically

```
class BasicBlob {  ………. }
class Blob extends BasicBlob { ……… }
public class MemoryCheck {
  public static void main(String[] args) {                  // (6)
    int blobsRequired, blobSize;
    try {
      blobsRequired = Integer.parseInt(args[0]);
      blobSize   = Integer.parseInt(args[1]);
    } catch(IndexOutOfBoundsException e) {
      System.err.println(
              "Usage: MemoryCheck <num of blobs> <blob size>");
      return;
    }
```

Invoking Garbage Collection Programmatically

```
Runtime environment = Runtime.getRuntime();                             // (7)
    System.out.println("Total memory: " + environment.totalMemory()); // (8)
    System.out.println("Free memory before blob creation: " +
      environment.freeMemory());                                        // (9)
    for (int i=0; i<blobsRequired; ++i) {                              // (10)
      new Blob(blobSize);
    }
    System.out.println("Free memory after blob creation: " +
      environment.freeMemory());                                       // (11)
    System.gc();                                                       // (12)
    System.out.println("Free memory after requesting GC: " +
      environment.freeMemory());                                       // (13)
    System.out.println(BasicBlob.population + " blobs alive");         // (14)
  }
}
```

Initializers

- Initializers can be used to set initial values for fields in objects and classes.
- There are three different types of initializers:
  - ➢field initializer expressions
  - ➢static initializer blocks
  - ➢instance initializer blocks

Field Initializer Expressions

- Initialization of fields can be specified in field declaration statements using initializer expressions. The value of the initializer expression must be assignment compatible to the declared field
- We distinguish between static and non-static field initializers.

```
class ConstantInitializers {
  int minAge = 12;                    // (1) Non-static
  static double pensionPoints = 10.5; // (2) Static
  // ...
}
```

Field Initializer Expressions

- When a class is loaded, it is initialized, i.e., its static fields are initialized with the values of the initializer expressions.
- The declaration at (2) will result in the static field `pensionPoints` being initialized to `10.5` when the class is loaded by the JVM.
- An initializer expression for a static field cannot refer to non-static members by their simple names.
- The keywords **this** and **super** cannot occur in a static initializer expression.

Field Initializer Expressions

- Since a class is always initialized before it can be instantiated, an instance initializer expression can always refer to any static member of a class, regardless of the member declaration order.
- In the following code, the instance initializer expression at (1) refers to the static field NO_OF_WEEKS declared and initialized at (2).
- Such a forward reference is legal. More examples of forward references are given in the next slides

Field Initializer Expressions

```
class MoreInitializers {
  int noOfDays = 7 * NO_OF_WEEKS;  // (1) Non-static
  static int NO_OF_WEEKS = 52;     // (2) Static
  // ...
}
```

Forward References and Declaration Order of Initializer Expressions

- When an object is created using the new operator, instance initializer expressions are executed in the order in which the instance fields are declared in the class.

```
class NonStaticInitializers {
  int length  = 10;                       // (1)
//double area = length * width;           // (2) Not Ok.
                                          // Illegal forward reference.
  double area = length * this.width;      // (3) Ok,
                                          //but width has default value 0.
  int width   = 10;                       // (4)
  int sqSide = height = 20;             // (5) OK. Legal forward reference.
  int height;                           // (6)
}
```

Initializer Expression Order and Method Calls

```
class Hotel {
  private int noOfRooms  = 12;                // (1)
  private int maxNoOfGuests    = initMaxGuests();          // (2)
Bug
  private int occupancyPerRoom = 2;                // (3)
  public int initMaxGuests() {                  // (4)
    System.out.println("occupancyPerRoom: " + occupancyPerRoom);
    System.out.println("maxNoOfGuests:    " + noOfRooms *
occupancyPerRoom);
    return noOfRooms * occupancyPerRoom;
  }
  public int getMaxGuests() { return maxNoOfGuests; }          //
(5)
  public int getOccupancy() { return occupancyPerRoom; }
// (6)
}
```

Initializer Expression Order and Method Calls

```
public class TestOrder {
  public static void main(String[] args) {
    Hotel hotel = new Hotel();                                     //(7)
    System.out.println("After object creation: ");
    System.out.println("occupancyPerRoom: " + hotel.getOccupancy());//(8)
    System.out.println("maxNoOfGuests:    " + hotel.getMaxGuests());//(9)
  }
}
```

Output from the program:

```
occupancyPerRoom: 0
maxNoOfGuests:    0
After object creation:
occupancyPerRoom: 2
maxNoOfGuests:    0
```

Exception Handling and Initializer Expressions

- Initializer expressions in named classes and interfaces must not result in any uncaught checked exception.
- If any checked exception is thrown during execution of an initializer expression, it must be caught and handled by code called from the initializer expression.
- This restriction does not apply to instance initializer expressions in anonymous classes.

Static Initializer Blocks

- Java allows static initializer blocks to be defined in a class. Although such blocks can include arbitrary code, they are primarily used for initializing static fields. The code in a static initializer block is executed only once, when the class is initialized.
- The syntax of a static initializer block comprises the keyword static followed by a local block that can contain arbitrary code, as shown at (3).

```
class StaticInitializers {
  final static int ROWS = 12, COLUMNS = 10;          // (1)
  static long[][] matrix = new long[ROWS][COLUMNS];  // (2)
  // ...
  static {                                      // (3) Static Initializer
    for (int i = 0; i < matrix.length; i++)
      for (int j = 0; j < matrix[i].length; j++)
        matrix[i][j] = 2*i + j;
  }
  // ...
}
```

Static Initializer Blocks

- If a class relies on native method implementations, a static initializer can be used to load any external libraries that the class needs.
- Static initializer block is not contained in any method. 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 because they cannot be called directly.

Forward References and Declaration Order of Static Initializers

```
public 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
  }
  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);
  }
}
```

Exception Handling and Static Initializer Blocks

- Exception handling in static initializer blocks is no different from that in static initializer expressions: uncaught checked exceptions cannot be thrown.
- A static initializer block cannot be called directly, therefore, any checked exceptions must be caught and handled in the body of the static initializer block.

```
class BankrupcyException
      extends RuntimeException {}      // (1) Unchecked Exception
class TooManyHotelsException
      extends Exception {}             // (2) Checked Exception
class Hotel {
  // Static Members
  private static boolean bankrupt   = true;
  private static int   noOfHotels = 11;
  private static Hotel[] hotelPool;
  static {                             // (3) Static block
    try {                              // (4) Handles checked exception
      if (noOfHotels > 10)
        throw new TooManyHotelsException();
    } catch (TooManyHotelsException e) {
      noOfHotels = 10;
      System.out.println("No. of hotels adjusted to " + noOfHotels);
    }
    hotelPool = new Hotel[noOfHotels];
  }
  static {                             // (5) Static block
    if (bankrupt)
      throw new BankrupcyException(); // (6) Throws unchecked exception
  }
  // ...
}
```

Exception Handling and Static Initializer Blocks

```
public class ExceptionInStaticInitBlocks {
  public static void main(String[] args) {
    new Hotel();
  }
}
```

Output from the program:

```
No. of hotels adjusted to 10
Exception in thread "main" java.lang.ExceptionInInitializerError
        at
ExceptionInStaticInitBlocks.main(ExceptionInStaticInitBlocks.java:32)
Caused by: BankrupcyException
        at Hotel.<clinit>(ExceptionInStaticInitBlocks.java:25)
```

Instance Initializer Blocks

- Just as static initializer blocks can be used to initialize static fields in a named class, Java provides the ability to initialize fields during object creation using instance initializer blocks.
- In this respect, such blocks serve the same purpose as constructors during object creation. The syntax of an instance initializer block is the same as that of a local block, as shown at (2) in the following code.
- The code in the local block is executed every time an instance of the class is created.

```
class InstanceInitializers {
  long[] squares = new long[10];    // (1)
  // ...
  {  // (2) Instance Initializer
    for (int i = 0; i < squares.length; i++)
      squares[i] = i*i;
  }
  // ...
}
```

Forward References and Declaration Order of Instance Initializers

- Analogous to other initializers discussed so far, an instance initializer block cannot make a forward reference to a field that violates the declaration-before-reading rule.
- In next example, an illegal forward reference occurs in the code at (4), which attempts to read the value of the field nsf1 before it is declared.
- The read operation at (11) is after the declaration and is, therefore, allowed.
- Forward reference made on the left-hand side of the assignment is always allowed, as shown at (2), (3), (5), and (7).

```
class NonStaticForwardReferences {
  {                // (1) Instance initializer block
    nsf1 = 10;     // (2) OK. Assignment to nsf1 allowed
    nsf1 = sf1;    // (3) OK. Static field access in non-static context
//  int a = 2 * nsf1; // (4) Not OK. Read operation before declaration
    int b = nsf1 = 20;  // (5) OK. Assignment to nsf1 allowed
    int c = this.nsf1; // (6) OK. Not accessed by simple name
  }
  int nsf1 = nsf2 = 30; // (7) Non-static field. Assignment to nsf2 allowed
  int nsf2;             // (8) Non-static field
  static int sf1 = 5;   // (9) Static field
  {       // (10) Instance initializer block
    int d = 2 * nsf1; // (11) OK. Read operation after declaration
    int e = nsf1 = 50;     // (12)
  }
  public static void main(String[] args) {
    NonStaticForwardReferences objRef = new NonStaticForwardReferences();
    System.out.println("nsf1: " + objRef.nsf1);
    System.out.println("nsf2: " + objRef.nsf2);
  }
}
```

Instance Initializer Blocks

- As in a instance initializer expression, the keywords this and super can be used to refer to the current object in an instance initializer block.
- As in a static initializer block, the return statement is also not allowed in instance initializer blocks.
- An instance initializer block can be used to factor out common initialization code that will be executed regardless of which constructor is invoked.
- A typical usage of an instance initializer block is in anonymous classes, which cannot declare constructors, but can instead use instance initializer blocks to initialize fields.

Exception Handling and Instance Initializer Blocks

- Exception handling in instance initializer blocks is similar to that in static initializer blocks.
- Exception handling in instance initializer blocks differs from that in static initializer blocks in the following aspect: 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.
- Static initializer blocks cannot allow this, since no constructors are involved in class initialization.
- Instance initializer blocks in anonymous classes have even greater freedom: they can throw any exception.

Constructing Initial Object State

- Object initialization involves constructing the initial state of an object when it is created by using the new operator.
- First, the fields are initialized to their default values—whether they are subsequently given non-default initial values or not—then the constructor is invoked.
- This can lead to local chaining of constructors.

Constructing Initial Object State

- The invocation of the constructor at the end of the local chain of constructor invocations results in the following actions, before the constructor’s execution resumes:
  - ➢ Implicit or explicit invocation of the superclass constructor. Constructor chaining ensures that the inherited state of the object is constructed first
  - ➢ Initialization of the instance fields by executing their instance initializer expressions and any instance initializer blocks, in the order they are specified in the class declaration

```java
class SuperclassA {
  public SuperclassA() {                             // (1)
    System.out.println("Constructor in SuperclassA");
  }
}
class SubclassB extends SuperclassA {
  SubclassB() {                                      // (2) Default constructor
    this(3);
    System.out.println("Default constructor in SubclassB");
  }
  SubclassB(int i) {                                 // (3) Non-default constructor
    System.out.println("Non-default constructor in SubclassB");
    value = i;
  }
  {    // (4) Instance initializer block
    System.out.println("Instance initializer block in SubclassB");
    value = 2;                          // (5)
  }
  int value = initializerExpression();          // (6)
  private int initializerExpression() {         // (7)
    System.out.println("Instance initializer expression in SubclassB");
    return 1;
  }
}
```

Constructing Initial Object State

```
public class ObjectConstruction {
  public static void main(String[] args) {
    SubclassB objRef = new SubclassB();          // (8)
    System.out.println("value: " + objRef.value);
  }
}
```

Output from the program:

```
Constructor in SuperclassA
Instance initializer block in SubclassB
Instance initializer expression in SubclassB
Non-default constructor in SubclassB
Default constructor in SubclassB
value: 3
```

```java
class SuperclassA {
  protected int superValue;                                 // (1)
  SuperclassA() {                                           // (2)
    System.out.println("Constructor in SuperclassA");
    this.doValue();                                         // (3)
  }
  void doValue() {                                          // (4)
    this.superValue = 911;
    System.out.println("superValue: " + this.superValue);
  }
}
class SubclassB extends SuperclassA {
  private int value = 800;                                  // (5)
  SubclassB() {                                             // (6)
    System.out.println("Constructor in SubclassB");
    this.doValue();
    System.out.println("superValue: " + this.superValue);
  }
  void doValue() {                                          // (7)
    System.out.println("value: " + this.value);
  }
}
```

Initialization under Object State Construction

```
public class ObjectInitialization {
  public static void main(String[] args) {
    System.out.println("Creating an object of SubclassB.");
    new SubclassB();                    // (8)
  }
}
```

Output from the program:

```
Creating an object of SubclassB.
Constructor in SuperclassA
value: 0
Constructor in SubclassB
value: 800
superValue: 0
```

That’s all