Unit 3

Topic: Multithreading and Exception Handling

Unit 3/Lecture 1

MultiThreading [RGPV/June 2011(10)]

Unlike many other computer languages, Java provides built-in support for *multithreaded programming.* A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: processbased and thread-based. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In processbased multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling.* In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. The benefit of Java’s multithreading is that the main loop/polling mechanism is eliminated.

One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run. Threads exist in several states. A thread can be *running.* It can be *ready to run* as soon as it gets CPU time. Arunning thread can be *suspended,* which temporarily suspends its activity.

Asuspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Creating a Thread

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

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

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method
called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable**

interface.
This defines where execution of the thread will begin. The name of the new thread is specified
by *threadName*. After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )**

method is shown here:
void start( )
Here is an example that creates a new thread and starts it running:
// Create a second thread.

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

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:
t = new Thread(this, "Demo Thread");
Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows. (Your output may vary based on processor speed and task load.)

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

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

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

```
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
```

```
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}


class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)
Here, *threadName* specifies the name of the thread.

Thread Priorities

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

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

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

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	In what situation a runnable interface is required to launch a thread?	Dec 2009	2
Q.2	What are the basic idea of multithreaded programming and its synchronization?	Dec 2009	10
Q-3	Thread Synchronization	Dec 2010	15
Q-4	What do you mean by Multithreading? Write a sample code for Multithreading?	June 2011	10

Unit 3

Topic: Thread Synchronization

Unit 3/Lecture 2

Thread Synchronization [RGPV/Dec-2009(10)][RGPV/Dec-2010(5)]

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

Synchronization in Java

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread .sleep(1000)**, which pauses the current thread for one second. The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
Class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
```

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

Here is the output produced by this program:

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

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition,* because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To

do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

class Callme {
synchronized void call(String msg) {
...
This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

[Hello]
[Synchronized]
[World]

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

Class Callme {
synchronized void call(String msg) {

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

[Hello]
[Synchronized]
[World]

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement [RGPV Dec 2014(7)]
While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

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

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

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.
Class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello"); Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");// wait for threads to end
try {
```

```
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
```

}Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Define Synchronization. Explain synchronization method & synchronization block.	Dec 2014	7
Q-1.	Thread Synchronization	Dec 2010	15

Unit 3

Topic: Thread Scheduling

Unit 3/Lecture 3

Thread Scheduling

In our introduction to how threads work, we introduced the **thread scheduler**, part of the OS (usually) that is responsible for sharing the available CPUs out between the various threads. How exactly the scheduler works depends on the individual platform, but various modern operating systems (notably Windows and Linux) use largely similar techniques that we'll describe here. We'll also mention some key 12ehavior12s between the platforms.

Note that we'll continue to talk about a single thread scheduler. On multiprocessor systems, there is generally some kind of scheduler *per processor*, which then need to be coordinated in some way. (On some systems, switching on different processors is staggered to avoid contention on shared scheduling tables.) Unless otherwise specified, we'll use the term *thread scheduler* to refer to this overall system of coordinated per-CPU schedulers.

Across platforms, thread scheduling[1] tends to be based on at least the following criteria:

- a **priority**, or in fact usually *multiple* "priority" settings that we'll discuss below;
- a **quantum**, or number of allocated **timeslices** of CPU, which essentially determines the amount of CPU time a thread is allotted before it is forced to yield the CPU to another thread of the same or lower priority (the system will keep track of the *remaining* quantum at any given time, plus its *default* quantum, which could depend on thread type and/or system configuration);
- a **state**, notably "runnable" vs "waiting";
- **metrics** about the 12ehavior of threads, such as recent CPU usage or the time since it last *ran* (i.e. had a share of CPU), or the fact that it has "just received an event it was waiting for".

Most systems use what we might dub **priority-based round-robin** scheduling to some extent. The general principles are:

- a thread of **higher priority** (which is a function of base and local priorities) will **preempt** a thread of lower priority;
- otherwise, threads of **equal priority** will essentially **take turns** at getting an allocated slice or **quantum** of CPU;
- there are a few extra "tweaks" to make things work.

States

Depending on the system, there are various **states** that a thread can be in. Probably the two most interesting are:

- **runnable**, which essentially means "ready to consume CPU"; being runnable is generally the **minimum** requirement for a thread to actually be scheduled on to a CPU.

- **waiting**, meaning that the thread currently cannot continue as it is waiting for a resource such as a lock or I/O, for memory to be paged in, for a signal from another thread, or simply for a period of time to elapse (*sleep*).

Producer-Consumer Problem [RGPV/Dec 2013(10)]

The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource. In the problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. These processes do not take turns accessing the buffer, they both work concurrently. Herein lies the problem. What happens if the producer tries to put an item into a full buffer? What happens if the consumer tries to take an item from an empty buffer?

The producer must first create a new widget.

1) Then, it checks to see if the buffer is full. If it is, the producer will put itself to sleep until the consumer wakes it up. A "wakeup" will come if the consumer finds the buffer empty.
2) Next, the producer puts the new widget in the buffer. If the producer goes to sleep in step (2), it will not wake up until the buffer is empty, so the buffer will never overflow.
3) Then, the producer checks to see if the buffer is empty. If it is, the producer assumes that the consumer is sleeping, an so it will wake the consumer. Keep in mind that between any of these steps, an interrupt might occur, allowing the consumer to run.

The consumer checks to see if the buffer is empty.

1). If so, the consumer will put itself to sleep until the producer wakes it up. A "wakeup" will occur if the producer finds the buffer empty after it puts an item into the buffer.

(2) Then, the consumer will remove a widget from the buffer. The consumer will never try to remove a widget from an empty buffer because it will not wake up until the buffer is full.

(3) If the buffer was full before it removed the widget, the consumer will wake the producer.

(4) Finally, the consumer will consume the widget. As was the case with the producer, an interrupt could occur between any of these steps, allowing the producer to run.

```
Import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to solve Producer Consumer problem using wait and notify
 * method in Java. Producer Consumer is also a popular concurrency design pattern.
 *
```

```
 * @author Javin Paul
 */
public class ProducerConsumerSolution {

   public static void main(String args[]) {
      Vector sharedQueue = new Vector();
      int size = 4;
      Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");
      Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");
      prodThread.start();
      consThread.start();
   }
}

class Producer implements Runnable {

   private final Vector sharedQueue;
   private final int SIZE;

   public Producer(Vector sharedQueue, int size) {
      this.sharedQueue = sharedQueue;
      this.SIZE = size;
   }

   @Override
   public void run() {
      for (int i = 0; i < 7; i++) {
         System.out.println("Produced: " + i);
         try {
            produce(i);
         } catch (InterruptedException ex) {
            Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
         }

      }
   }

   private void produce(int i) throws InterruptedException {

      //wait if queue is full
      while (sharedQueue.size() == SIZE) {
         synchronized (sharedQueue) {
            System.out.println("Queue is full " + Thread.currentThread().getName()
                        + " is waiting , size: " + sharedQueue.size());

            sharedQueue.wait();
         }
```

```
        }

        //producing element and notify consumers
        synchronized (sharedQueue) {
            sharedQueue.add(i);
            sharedQueue.notifyAll();
        }
    }
}

class Consumer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumed: " + consume());
                Thread.sleep(50);
            } catch (InterruptedException ex) {
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);
            }

        }
    }

    private int consume() throws InterruptedException {
        //wait if queue is empty
        while (sharedQueue.isEmpty()) {
            synchronized (sharedQueue) {
                System.out.println("Queue is empty " + Thread.currentThread().getName()
                                + " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //Otherwise consume element and notify waiting producer
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();
```

```
        return (Integer) sharedQueue.remove(0);
      }
    }
}
```

Output:

```
Produced: 0
Queue is empty Consumer is waiting , size: 0
Produced: 1
Consumed: 0
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Queue is full Producer is waiting , size: 4
Consumed: 1
Produced: 6
Queue is full Producer is waiting , size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Queue is empty Consumer is waiting , size: 0
```

Daemon Thread[RGPV/Dec-2010(5) Dec 2014(2)]]

A "daemon" thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus, when all of the non-daemon threads complete, the program is terminated. Conversely, if there are any non-daemon threads still running, the program doesn't terminate. There is, for instance, a non-daemon thread that runs **main( )**.

```
//: c13:SimpleDaemons.java
// Daemon threads don't prevent the program from ending.

Public class SimpleDaemons extends Thread {
  public SimpleDaemons() {
    setDaemon(true); // Must be called before start()
    start();
  }
  public void run() {
    while(true) {
      try {
        sleep(100);
      } catch (InterruptedException e) {
```

```
      throw new RuntimeException(e);
    }
    System.out.println(this);
  }
 }
 public static void main(String[] args) {
  for(int i = 0; i < 10; i++)
    new SimpleDaemons();
 }
} ///:~
```

You must set the thread to be a daemon by calling **setDaemon( )** before it is started. In **run( )**, the thread is put to sleep for a little bit. Once the threads are all started, the program terminates immediately, before any threads can print themselves, because there are no non-daemon threads (other than **main( )**) holding the program open. Thus, the program terminates without printing any output.

You can find out if a thread is a daemon by calling **isDaemon( )**. If a thread is a daemon, then any threads it creates will automatically be daemons, as the following example demonstrates:

```
//: c13:Daemons.java
// Daemon threads spawn other daemon threads.
Import java.io.*;
import com.bruceeckel.simpletest.*;

class Daemon extends Thread {
 private Thread[] t = new Thread[10];
 public Daemon() {
  setDaemon(true);
  start();
 }
 public void run() {
  for(int i = 0; i < t.length; i++)
    t[i] = new DaemonSpawn(i);
  for(int i = 0; i < t.length; i++)
    System.out.println("t[" + i + "].isDaemon() = "
      + t[i].isDaemon());
  while(true)
    yield();
 }
}

class DaemonSpawn extends Thread {
 public DaemonSpawn(int i) {
  start();
  System.out.println("DaemonSpawn " + i + " started");
```

```
  }
  public void run() {
    while(true)
      yield();
  }
}

public class Daemons {
  private static Test monitor = new Test();
  public static void main(String[] args) throws Exception {
    Thread d = new Daemon();
    System.out.println("d.isDaemon() = " + d.isDaemon());
    // Allow the daemon threads to
    // finish their startup processes:
    Thread.sleep(1000);
    monitor.expect(new String[] {
      "d.isDaemon() = true",
      "DaemonSpawn 0 started",
      "DaemonSpawn 1 started",
      "DaemonSpawn 2 started",
      "DaemonSpawn 3 started",
      "DaemonSpawn 4 started",
      "DaemonSpawn 5 started",
      "DaemonSpawn 6 started",
      "DaemonSpawn 7 started",
      "DaemonSpawn 8 started",
      "DaemonSpawn 9 started",
      "t[0].isDaemon() = true",
      "t[1].isDaemon() = true",
      "t[2].isDaemon() = true",
      "t[3].isDaemon() = true",
      "t[4].isDaemon() = true",
      "t[5].isDaemon() = true",
      "t[6].isDaemon() = true",
      "t[7].isDaemon() = true",
      "t[8].isDaemon() = true",
      "t[9].isDaemon() = true"
    }, Test.IGNORE_ORDER + Test.WAIT);
  }
} ///:~
```

The **Daemon** thread sets its daemon flag to "true" and then spawns a bunch of other threads—which *do not* set themselves to daemon mode—to show that they are daemons anyway. Then it goes into an infinite loop that calls **yield( )** to give up control to the other processes.

There's nothing to keep the program from terminating once **main( )** finishes its job, since there are

nothing but daemon threads running. So that you can see the results of starting all the daemon threads, the **main( )** thread is put to sleep for a second. Without this, you see only some of the results from the creation of the daemon threads. (Try **sleep( )** calls of various lengths to see this behavior.)

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is meant by Deamon Thread?	Dec 2014	2
Q.2	Explain the Producer Consumer Problem and demonstrate with the help of program. How this problem can be solved by using wait() and notify() Methods.	Dec 2013	10
Q-3	Explain the Daemon Thread in brief.	Dec-2010	5

Unit 3
Topic: Exception Handling
Unit 3/Lecture 4

Exception Handling [RGPV/Dec 2009(10)]

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

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

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

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

Here, *ExceptionType* is the type of exception that has occurred.

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to

stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

Using Try and Catch
Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped. running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
```

```
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:
Division by zero.
After catch statement.

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-byzero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	What is an Exception? Explain with example, how exceptions are handled in Java?	Dec 2010	05
Q-2.	Explain Exception Handling in Java.	Dec-2012	10

Unit 3

Topic: Multiple Catch Clauses

Unit 3/ Lecture 5

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

This program will cause a division-by-zero exception if it is started with no commandline arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to c[42].

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a

subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

Throw Statement[RGPV Dec 2014(7)]

However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause, or creating one with the **new** operator. The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

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

This program gets two chances to deal with the same error. First, **main( )** sets up an exception

context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exceptionhandling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting

output:
Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line.

Throws Statement

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A**throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a **throws** clause:

type method-name(*parameter-list*) throws *exception-list*
{
// body of method
}

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
```

```
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Finally Statement[RGPV Dec 2014(7)]

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

finally creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
```

```
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.


static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement
executes normally, without error. However, the **finally** block is still executed. ***REMEMBER*** *If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.*

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Define throw & throws in exception handling. Write a simple code to throw and handle an exception.	Dec 2014	7
Q.1	Explain the Term Try, Catch and Throw in Java.	Dec 2009	10
Q-2.	Create a Block that is likely to generate three types of exceptions and incorporate necessary catch blocks to catch and handle them appropriately.	Dec 2010	12

Unit 3

Topic: Stack-based Execution and Exception Propagation

Unit 3/ Lecture 6

Stack-based Execution and Exception Propagation

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, or an array index is out of bounds, or a network link failed. Explicit checks in the code for such conditions can easily result in incomprehensible code. Java provides an exception handling mechanism for systematically dealing with such error conditions. The exception mechanism is built around the throw-and-catch paradigm. To throw an exception is to signal that an unexpected error condition has occurred. To catch an exception is to take appropriate action to deal with the exception. An exception is caught by an exception handler, and the exception need not be caught in the same context that it was thrown in. The runtime behavior of the program determines which exceptions are thrown and how they are caught. The throw-and-catch principle is embedded in the

try
-
catch
-
finally

construct. Several threads can be executing in the JVM. Each thread has its own runtime stack(also called the call stack or the invocation stack) that is used to handle execution of methods. Each element on the stack (called an activation record or a stack frame) corresponds to a method call. Eachnew call results in a new activation record being pushed on the stack, which stores all the pertinent information such as storage for the local variables. The method with the activation record on top of the stack is the one currently executing. When this method finishes executing, its record is popped from the stack. Execution then continues in the method corresponding to the activation record which is now uncovered on top of the stack. The methods on the stack are said to be active , as their execution has not completed. At any given time, the active methods on a runtime stack comprise what is called the stack trace of a thread's execution.

The example given below shows a simple program to illustrate method execution. It calculates the average for a list of integers, given the sum of all the integers and the number of integers. It uses three methods:

- The method main() which calls the method printAverage() with parameters giving the total sum of the integers and the total number of integers, (1).
- The method printAverage() in its turn calls the method computeAverage() , (3).
- The method computeAverage() uses integer division to calculate the average and returns the result, (7).

Example
Method Execution

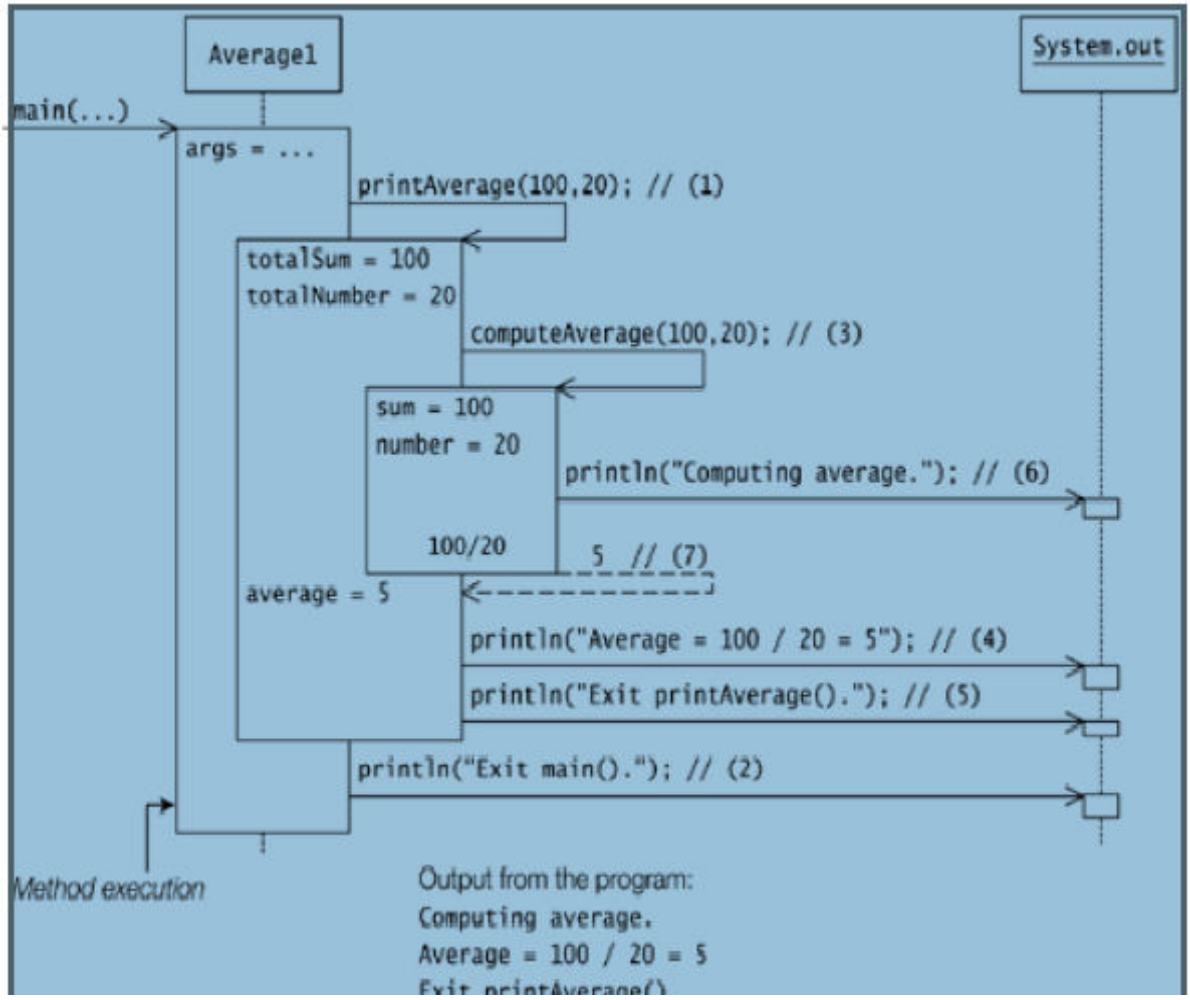
```
public class Average1 {
public static void main(String[] args) {
printAverage(100,0); // (1)
System.out.println("Exit main()."); // (2)
}
public static void printAverage(int totalSum, int totalNumber) {
int average = computeAverage(totalSum, totalNumber); // (3)
System.out.println("Average = " + // (4)
totalSum + " / " + totalNumber + " = " + average);
System.out.println("Exit printAverage()."); // (5)
}
public static int computeAverage(int sum, int number) {
System.out.println("Computing average."); // (6)
return sum/number; // (7)
}
}
```

Output of program execution:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Execution of above example is illustrated in Figure given below. Each method execution is shown as a box with the local variables. The box height indicates how long a method is active. Before the call to the method System.out.println()at (6) in Figure given below, the stack trace comprises of the three active methods:

main(),printAverage()and computeAverage(). The result 5 from the method computeAverage() is returned at (7) in Figure given below. The output from the program is in correspondence with the sequence of method calls in Figure given below.



If the method call at (1) in above Example

printAverage(100, 20); // (1)

is replaced with

printAverage(100, 0); // (1)

and the program is run again, the output is as follows:
Computing average.

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Average1.computeAverage(Average1.java:18)
at Average1.printAverage(Average1.java:10)
at Average1.main(Average1.java:5)

The Figure given below illustrates the program execution. All goes well until the return statement at (7) in the method computeAverage () is executed. An error condition occurs in calculating the expression sum/number, because integer division by 0 is an illegal operation.

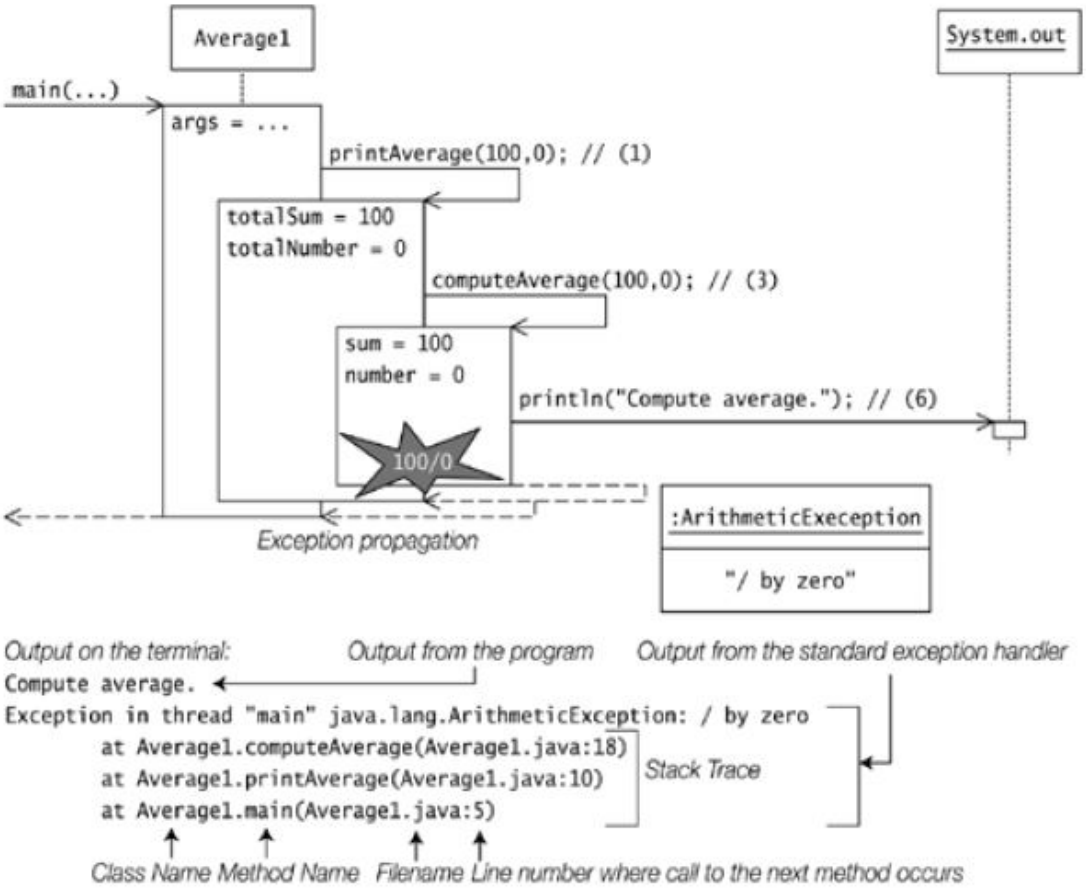


Fig. 3.1: Exception Propagation

The above figure illustrates the case where an exception is thrown and the program does not take any explicit action to deal with the exception. In aboe Figure, execution of the computeAverage()method is stopped at the point where the exception is thrown. The execution of the return statement at (7) never gets completed. Since this method does not have any code to deal with the exception, its execution is likewise terminated abruptly and its activation record popped. We say that the method completes abruptly. The exception is then offered to the method whose activation is now on top of the stack (method printAverage ()). This method does not have any code to deal with the exception either, so its execution completes abruptly. Lines (4) and (5) in the method printAverage () never get executed. The exception now propagates to the last active method (method main ()). This does not deal with the exception either. The main() method also completes abruptly. Line (2) in the main () method never gets executed. Since the exception is not caught by any of the active methods, it is dealt with by the main thread's default exception handler. The default exception handler usually prints the name of the exception, with an explanatory message, followed by a printout of the stack trace at the time the exception was thrown. An uncaught exception results in the death of the thread in which the exception occurred. If an exception is thrown during the evaluation of the left-hand operand of a binary

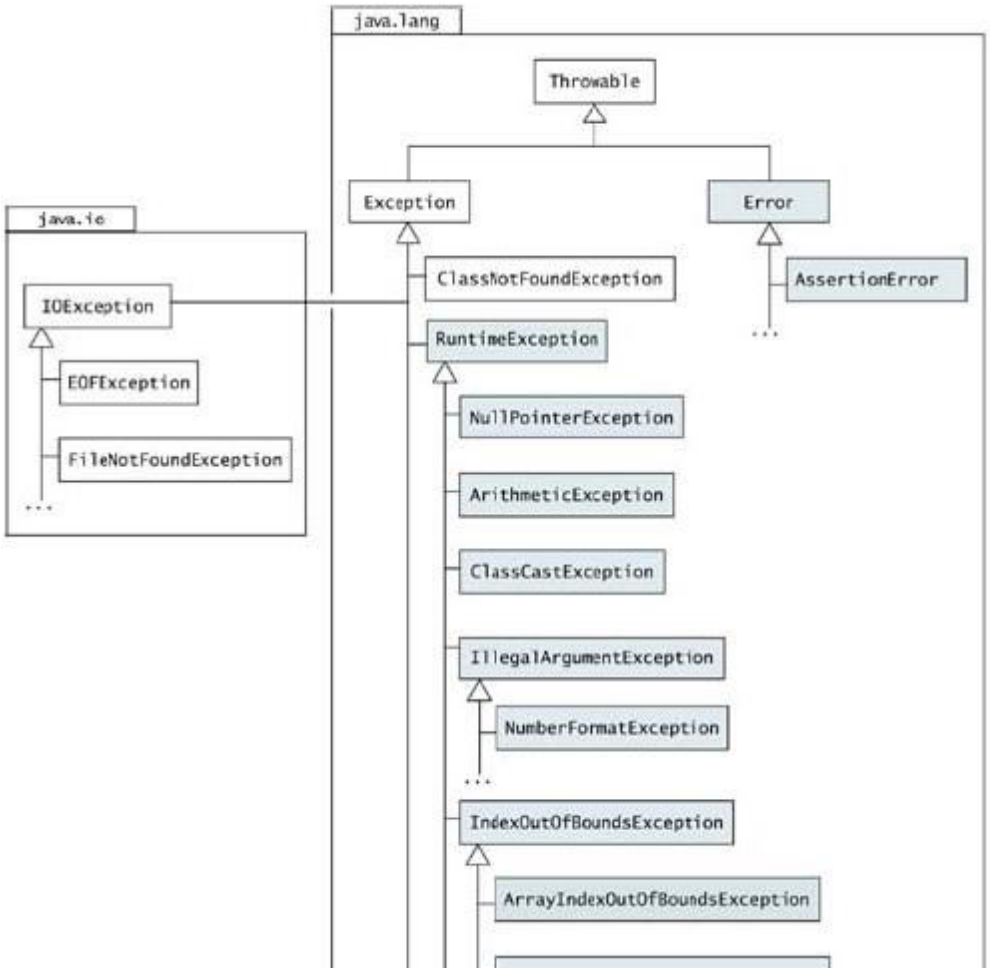
expression, then the right operand is not evaluated. Similarly if an exception is thrown during the evaluation of a list of expressions (for example, a list of actual parameters in a method call), then evaluation of the rest of the list is skipped.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the difference between final, finally & finalize?	Dec 2014	3

Unit 3
Topic: Exception Types
Unit 3/Lecture 7

Exceptions [RGPV Dec 2014(7)]

Exception in Java are objects. All exceptions are derived from the java.lang. Throwable class. Figure 5.8 shows a partial hierarchy of classes derived from the Throwable class. The two main subclasses Exception and Error constitute the main categories of throwables, the term used to refer to both exceptions and errors. Figure 5.8 also shows that not all exception classes are found in the same package.



The Throwable class provides a String variable that can be set by the subclasses to provide a detail message. The purpose of the detail message is to provide more information about the actual exception. All classes of throwables define a one-parameter constructor that takes a string as the detail message. 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. Its subclass RuntimeException represents many common programming errors that manifest at runtime (see the next subsection). Other subclasses of the Exception class define other categories of exceptions, for example, I/O-related exceptions (IOException, FileNotFoundException, EOFException) and GUI-related exceptions (AWTException).

Class RuntimeException

Runtime exceptions, like out-of-bound array indices (ArrayIndexOutOfBounds Exception),uninitialized references (NullPointerException), illegal casting of references(ClassCastException), illegal parameters (IllegalArgumentException), division by zero (ArithmeticException), and number format problems (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. Other subclasses of the java.lang.Error class define exceptions that indicate class linkage (LinkageError), thread (ThreadDeath), and virtual machine (VirtualMachineError) related problems. 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 (see Exceptions defined by Errorand RuntimeException classes and their subclasses are known as Unchecked exceptions, meaning that a method is not obliged to deal with these kinds of exceptions (shown with grey color in Figure 5.8). They are either irrecoverable (exemplified by the Error class) and the program should not attempt to deal with them, or they are programming errors (exemplified by the RuntimeException class) and should be dealt with as such and not as exceptions.

Defining New Exceptions 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. 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. Note that the exception class must be instantiated to create an exception object that can be thrown and subsequently caught and dealt with. The code below sketches a class definition for an exception that can include all pertinent information about the exception.

```
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
// ...
}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1	Explain various types of exception in Java.	June 2011	10

Unit 3
Topic: Assertions
Unit 3/ Lecture 8

Assertions

Assertions in Java can be used to document and validate assumptions made about the state of the program at designated locations in the code. Each assertion contains a boolean expression that is expected to be true when the assertion is executed. If this assumption is false, the system throws a special assertion error. The assertion facility uses the exception handling mechanism to propagate the error. The assertion facility can be enabled or disabled at runtime. The assertion facility is an invaluable aid in implementing correct programs (i.e., programs that adhere to their specification). It should not be confused with the exception handling mechanism that aids in developing robust programs (i.e., programs that handle unexpected conditions gracefully). Used judiciously, the two mechanisms facilitate programs that are reliable

Assert Statement and AssertionError Class

The following two forms of the assert statement can be used to specify assertios:
assert <boolean expression> ; // the simple form
assert <boolean expression> :<message expression>
; // the augmented form

If assertions are enabled (see p. 212), the execution of an assert statement proceeds as shown in Figure 5.13 . The two forms are essentially equivalent to the following code, respectively:

if (<assertions enabled> && !<boolean expression>) // the simple form throw new AssertionError();
if (<assertions enabled>&& ! <boolean expression>) // the augmented form throw new AssertionError(<message expression>
);

Example
Assertions

```
public class Speed {
public static void main(String[] args) {
Speed objRef = new Speed();
double speed = objRef.calcSpeed(-12.0, 3.0); // (1a)
// double speed = objRef.calcSpeed(12.0, -3.0); // (1b)
// double speed = objRef.calcSpeed(12.0, 2.0); // (1c)
// double speed = objRef.calcSpeed(12.0, 0.0); // (1d)
System.out.println("Speed (km/h): " + speed);
}
/** Requires distance >= 0.0 and time > 0.0 */
```

```
private double calcSpeed(double distance, double time) {
assert distance >= 0.0; // (2)
assert time >0.0 : "Time is not a positive value: " + time; // (3)
double speed = distance / time;
assert speed >= 0.0; // (4)
return speed;
}
}
```

Compiling Assertions

The assertion facility was introduced in J2SE 1.4. At the same time, two new options for the Javac compiler were introduced for dealing with assertions in the source code. Option -source 1.4 The javac compiler distributed with the Java SDK v1.4 will only compile assertions if the option -source 1.4 is used on the command-line:

```
>javac -source 1.4 Speed.java
```

This also means that incorrect use of the keyword assert will be flagged as an error , for example, if assert is used as an identifier.

References:

Book	Authr	Priority
Java Programming	Herbertt Schield	1
Java	E Balaguruswamy	2
Java	Khalid Mugal	3