Exceptions in Java

Java provides a fairly elaborate exception handling mechanism based on the throw-catch model.

All exceptions are objects, required to be a subclass of `Throwable`.

Class `Throwable` has two subclasses, `Exception` and `Error`. Class `Exception` has a subclass `RuntimeException`.

Exceptions may be explicitly thrown (using a `throw` statement) or they may be implicitly thrown as the result of a run-time error.

For example, an `ArithmeticException` is thrown for certain run-time arithmetic errors, like division by zero.

Unlike other languages, Java divides exceptions into two general classes: ***checked*** **and** ***unchecked*****.**

A checked exception ***must*** **either be caught (using a try-catch block) or propagated (by marking a method as throwing the exception).**

This means that checked exceptions cannot be ignored—you must be prepared to catch them or you must "advertise" to your callers that you may throw an exception back to them.

Unchecked exceptions need not be caught or marked as potentially thrown. This makes exception handling for such exceptions optional. Unchecked exceptions are typically those that might occur so

often (like `NullPointerException` or `ArithmeticException`) that forced checks could unnecessarily clutter a program without significant benefit.

How are checked and unchecked exceptions distinguished?

- Any exception that is a member (or subclass) of `Error` or `RuntimeException` is unchecked.
- All other exceptions must be checked.

Exceptions are propagated dynamically:

- When an exception is thrown (explicitly or implicitly) the innermost try-catch block that can "catch" the exception is selected, and the catch block that matches the exception is executed.

- A catch block "catches" a given exception if the class of the exception is the same as the class used in the catch. An exception that is a subclass of the catch's exception class will also be caught.
  Thus an catch that handles class `Throwable` catches *all* exceptions.
- If no catch can handle the exception in the current method, a return to the method's caller is forced, and the exception is rethrown from the point of call.
- This process is repeated until a catch that can handle the exception is found or until we force a return from the main method.

- If a return from the main method is forced, no handler exists. A run-time error message is printed ("Uncaught exception") and execution is terminated.
- One of the limitations of Java's exception mechanism (and similar mechanisms found in other languages) is that there is no "retry" mechanism. Once an exception is thrown, we never go back to the point where the exception occurred. This is why Scheme's call/cc mechanism is considered so special and unique.

Example:

```
class badValue extends Exception{
  float value;
  badValue(float f) {value=f;} }

float sqrt(float val)
    throws badValue {
  if (val < 0.0)
     throw new badValue(val);
  else return
        (float) Math.sqrt(val); }

try {
  System.out.println(
    "Ans = " + sqrt(-123.0));
} catch (badValue b) {
  System.out.println(
    "Can't take sqrt of "+b)
}
```

Reading Assignment

- Pizza Tutorial
  (linked from class web page)

Threads and Parallelism in Java

Java is one of the few “main stream” programming languages to explicitly provide for user-programmed parallelism in the form of threads.

A Java programmer may organize a program as several threads that may execute concurrently.

Even if the program is run on a uni-processor, use of threads may improve performance. This is because the threads can be multi-programmed, with threads switched automatically on I/O delays, page faults or even cache misses.

A program that is designed to support multiple threads is also “prepared” for future upgrades to multiprocessors or multi-threaded processors.

Java Threads

In Java any class that implements the **`Runnable`** interface can be started as a concurrent thread:

```
interface Runnable {
     public void run();
 }
```

When the thread is started, the method **`run`** begins to execute (perhaps concurrently with other threads of the main program).

You create a thread using the **`Thread`** constructor:

```
new Thread(RunnableObject)
```

Creating a thread does not start it; you must execute the **`start`** method within the **`Thread`** object.

When this is done, the `run` method immediately starts, and continues until that method terminates normally, or it throws an uncaught exception, or it is explicitly stopped, or the main program stops.

On uniprocessors, thread execution is interleaved; on multiprocessors or multithreaded architectures execution can be concurrent.

```
class DoSort implements Runnable {
 int [] data;
 DoSort(int[] in) {data=in;}
 public void run(){
   // sort the data array;
 }
}
```

```
class Test {
 public static void
    main(String args[]){
  DoSort d =
     new DoSort(new int[1000]);
  Thread t1 = new Thread(d);
  t1.start();
  // We can continue while t1
  // does its sort
 }
}
```

We can start multiple threads, and the threads can use **`sleep`** to delay or synchronize their execution:

```
class PingPong
    implements Runnable {
 int delay; String word;
 PingPong(String s,int i){
    delay=i;word=s;};
 public void run(){
   try {
    while(true){
      System.out.print(word+" ");
      Thread.sleep(delay);
   }}
   catch (InterruptedException e)
    {}
  }
```

```
public static void
   main(String args[]){
  Thread t1 =  new Thread(
    new PingPong("ping",33));
  Thread t2 = new Thread(
    new PingPong("PONG",100));
  t1.start();
  t2.start();
 }
}
```

```
ping PONG ping ping PONG ping
ping ping PONG ping ping PONG
ping ...
```

Synchronization in Java

We often want threads to co-operate, typically in how they access shared data structures.

Since thread execution is asynchronous, the details of how threads interact can be unpredictable.

Consider a method

```
 update() {
    n = n+1;
    val = f(n);
}
```

that updates fields of an object.

If two or more threads execute `update` concurrently, we might get unexpected or even illegal behavior.

(Why?)

A Java method may be *synchronized*, which guarantees that at most one thread can execute the method at a time. Other threads wishing access, are forced to wait until the currently executing thread completes.

Thus

```
void synchronized update() { ... }
```

can safely be used to update an object, even if multiple threads are active.

There is also a **synchronized** statement in Java that forces threads to execute a block of code sequentially.

```
synchronized(obj) {
   obj.n = obj.n+1;
   obj.val = f(obj.n);
}
```

Synchronization Primitives

The following operations are provided to allow threads to safely interact:

`wait()`	Sleep until awakened
`wait(n)`	Sleep until awakened or until **`n`** milliseconds pass
`notify()`	Wake up one sleeping thread
`notifyAll()`	Wake up all sleeping threads

Using these primitives, correct concurrent access to a shared data structure can be programed.

Consider a `Buffer` class in which independent threads may try to store or fetch data objects:

```
class Buffer {
  private Queue q;
  Buffer() { q = new Queue(); }
  public synchronized void
  put(Object obj) {
    q.enqueue(obj);
    notify();//Why is this needed?
  }
  public synchronized Object
  get() {
    while (q.isEmpty()) {
     //Why a while loop?
     wait();
    }
    return q.dequeue();
  }
}
```