Advanced Flow of Control -- Introduction

- Two additional mechanisms for controlling process execution are exceptions and threads

- Chapter 14 focuses on:
  - exception processing
  - catching and handling exceptions
  - creating new exceptions
  - separate process threads
  - synchronizing threads

Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program
- A program can therefore be separated into a normal execution flow and an *exception execution flow*
- An *error* is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught

Exception Handling

- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- The manner in which an exception is processed is an important design consideration

Exception Handling

- If an exception is ignored by the program, the program will terminate and produce an appropriate message
- The message includes a *call stack trace* that indicates on which line the exception occurred
- The call stack trace also shows the method call trail that lead to the execution of the offending line
- See `Zero.java`

The `try` Statement

- To process an exception when it occurs, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses, which contain code to process an exception
- Each catch clause has an associated exception type
- When an exception occurs, processing continues at the first catch clause that matches the exception type
- See `Adding.java`

Exception Propagation

- If it is not appropriate to handle the exception where it occurs, it can be handled at a higher level
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the outermost level
- A try block that contains a call to a method in which an exception is thrown can be used to catch that exception
- See `Propagation_Demo.java`

Exceptions

- An exception is either *checked* or *unchecked*
- A checked exception can only be thrown within a try block or within a method that is designated to throw that exception
- The compiler will complain if a checked exception is not handled appropriately
- An unchecked exception does not require explicit handling, though it could be processed that way

The `throw` Statement

- A programmer can define an exception by extending the appropriate class
- Exceptions are thrown using the throw statement:

    `throw` *exception-object*`;`

- See `Throw_Demo.java`
- Usually a throw statement is nested inside an if statement that evaluates the condition to see if the exception should be thrown

The `finally` Clause

- A try statement can have an optional clause designated by the reserved word `finally`
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- Also, if an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

Threads

- Processing can be broken into several separate *threads of control* which execute at the same time
- "At the same time" could mean true parallelism or simply interlaced concurrent processing
- A thread is one sequential flow of execution that occurs at the same time another sequential flow of execution is processing the same program
- They are not necessarily executing the same statements at the same time

Threads

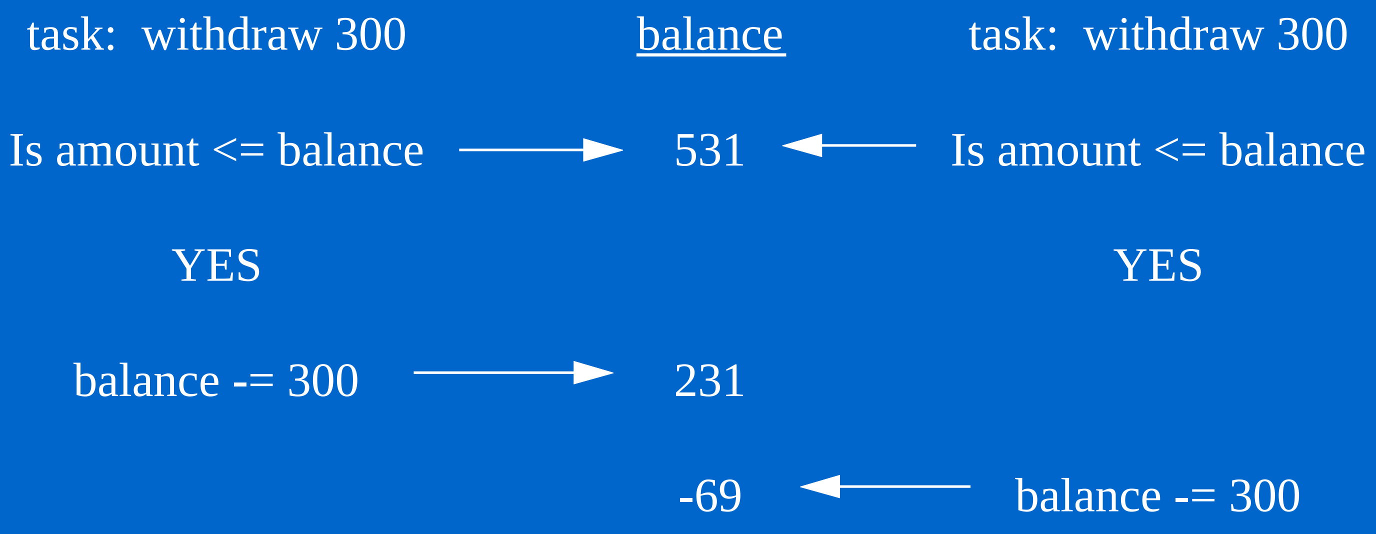
- A thread can be created by deriving a new thread from the `Thread` class
- The `run` method of the thread defines the concurrent activity, but the `start` method is used to begin the separate thread process
- See `Simultaneous.java`
- A thread can also be created by defining a class that implements the `Runnable` interface
- By implementing the interface, the thread class can be derived from a class other than `Thread`

Shared Data

- Potential problems arise when multiple threads share data
- Specific code of a thread may execute at any point relative to the processing of another thread
- If that code updates or references the shared data, unintended processing sequences can occur that result in incorrect results

Shared Data

- Consider two withdrawals from the same bank account at the same time



Synchronization

- Multiple threads of control can be made safe if areas of code that use shared data are *synchronized*
- When a set of code is synchronized, then only one thread can be using that code at a time
- The other threads must wait until the first thread is complete
- This is an implementation of a synchronization mechanism called a *monitor*
- See `ATM_Accounts.java`

Controlling Threads

- Thread processing can be temporarily suspended, then later resumed, using methods from the `Thread` class
- A thread can also be put to sleep for a specific amount of time
- These mechanisms can be quite helpful in certain situations, like controlling animations
- See `Bouncing_Ball2.java`