

Exceptions and Thread

Yao, Zhiyang

Example

```
import java.io.*;

public class A {
    // main(): application entry point
    public static void main(String[] args) throws IOException {
        // get filename
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.print("Filename: ");
        String s = stdin.readLine();

        // set up file stream for processing
        BufferedReader filein = new BufferedReader(new FileReader(s));

        // extract values and compute quotient
        int numerator = Integer.parseInt(filein.readLine());
        int denominator = Integer.parseInt(filein.readLine());

        int quotient = numerator / denominator;
        System.out.println();
        System.out.println(numerator + " / " + denominator + " = "
            + quotient);

        //return;
    }
}
```

Exception

- ❑ Abnormal event occurring during program execution
- ❑ Examples
 - Manipulate nonexistent files
`BufferedReader filein = new
BufferedReader(new FileReader(s));`
 - Improper array subscripting
`int[] a = new int[3];
a[4] = 1000;`
 - Improper arithmetic operations
`a[2] = 1000 / 0;`

Java treatment of an exception

- ❑ If exception occurs and a *handler* is in effect
 - Flow of control is transferred to the handler
 - After handler completes flow of control continues with the statement following the handler
- ❑ If exception occurs and there is no handler for it
 - The program terminates

Exception handlers

- Code that might generate an exception is put in a try block
 - If there is no exception, then the handlers are ignored
- For each potential exception type there is a catch handler
 - When handler finishes the program continues with statement after the handlers

```
try {  
    Code that might throw exceptions of types E or F  
}  
catch (E e) {  
    Handle exception e  
}  
catch (F f) {  
    Handle exception f  
}  
More code
```

Introduce try-catch blocks

```
public class D {  
    // main(): application entry point  
    public static void main(String[] args) {  
        // get filename  
        BufferedReader stdin = new BufferedReader(  
            new InputStreamReader(System.in));  
        System.out.print("Filename: ");  
        String s = null;  
        try {  
            s = stdin.readLine();  
        }  
        catch (IOException e) {  
            System.err.println("Cannot read input");  
            System.exit(0);  
        }  
  
        // set up file stream for processing  
        BufferedReader filein = null;  
        try {  
            filein = new BufferedReader(new FileReader(s));  
        }  
        catch (FileNotFoundException e) {  
            System.err.println(s + ": cannot be opened for reading");  
            System.exit(0);  
        }  
    }  
}
```

```
// extract values and compute quotient  
try {  
    int numerator = Integer.parseInt(filein.readLine());  
    int denominator = Integer.parseInt(filein.readLine());  
  
    int quotient = numerator / denominator;  
    System.out.println();  
    System.out.println(numerator + " / " + denominator + " = "  
        + quotient);  
}  
catch (IOException e) {  
    System.err.println(s + ": unable to read values");  
    System.exit(0);  
}  
catch (NumberFormatException e) {  
    if (e.getMessage().equals("null")) {  
        System.err.println(s + ": doesn't contain two inputs");  
    }  
    else {  
        System.err.println(s + ": contains nonnumeric inputs");  
    }  
    System.exit(0);  
}  
catch (ArithmeticException e) {  
    System.err.println(s + ": unexpected 0 input value");  
    System.exit(0);  
}  
return;  
}
```

Run time exceptions

- Java designers realized
 - Runtime exceptions can occur throughout a program
 - Cost of implementing handlers for runtime exceptions typically exceeds the expected benefit
- Java makes it optional for a method to catch them or to specify that it throws them
- However, if a program does not handle its runtime exceptions it is terminated when one occurs

Open a file

```
try {
    String s = filein.readLine();
    while (s != null) {
        System.out.println(s);
        s = filein.readLine();
    }
}
catch (IOException e) {
    System.err.println(args[i] + ": processing error");
}
```

What can go wrong?

File should be closed once its processing stops, regardless why it stopped

Use a finally block

```
try {
    String s = filein.readLine();
    while (s != null) {
        System.out.println(s);
        s = filein.readLine();
    }
}
catch (IOException e) {
    System.err.println(args[i] + ": processing error");
}
finally { ← Always executed after its try-catch blocks complete
    try {
        filein.close();
    }
    catch (IOException e) {
        System.err.println(args[i] + ": system error");
    }
}
```

Exceptions

- ❑ Can create your exception types
- ❑ Can throw exceptions as warranted

Task

- ❑ Represent the depositing and withdrawing of money from a bank account
- ❑ What behaviors are needed
 - Construct a new empty bank account
`BankAccount()`
 - Construct a new bank account with initial funds
`BankAccount(int n)`
 - Deposit funds
`addFunds(int n)`
 - Withdraw funds
`removeFunds(int n)`
 - Get balance
`int getBalance()`

What can go wrong?

Create a NegativeAmountException

// Represents an abnormal bank account event

```
public class NegativeAmountException extends Exception
{
    // NegativeAmountException(): creates exception with
    // message s
    public NegativeAmountException(String s) {
        super(s);
    }
}
```

- ❑ Class Exception provides the exceptions behavior that might be needed
- ❑ Class NegativeAmountException gives the specialization of exception type that is needed

Class BankAccount

- ❑ Instance variable
 balance
- ❑ Construct a new empty bank account
 BankAccount()
- ❑ Construct a new bank account with initial funds
 BankAccount(int n) throws NegativeAmountException
- ❑ Deposit funds
 addFunds(int n) throws NegativeAmountException
- ❑ Withdraw funds
 removeFunds(int n) throws NegativeAmountException
- ❑ Get balance
 int getBalance()

Class BankAccount

// BankAccount(): default constructor for empty balance

```
public BankAccount() {
    balance = 0;
}
```

// BankAccount(): specific constructor for a new balance n

```
public BankAccount(int n) throws
    NegativeAmountException {
    if (n >= 0) {
        balance = n;
    }
    else {
        throw new NegativeAmountException("Bad balance");
    }
}
```

Class BankAccount

// getBalance(): return the current balance

```
public int getBalance() {
    return balance;
}
```

// addFunds(): deposit amount n

```
public void addFunds(int n) throws
    NegativeAmountException {
    if (n >= 0) {
        balance += n;
    }
    else {
        throw new NegativeAmountException("Bad deposit");
    }
}
```

Class BankAccount

```
// removeFunds(): withdraw amount n
public void removeFunds(int n) throws
    NegativeAmountException {
    if (n < 0) {
        throw new NegativeAmountException("Bad
        withdrawal");
    }
    else if (balance < n) {
        throw new NegativeAmountException("Bad balance");
    }
    else {
        balance -= n;
    }
}
```


Threads

Story so far

- Our programs have consisted of single flows of control
 - Flow of control started in the first statement of method main() and worked its way statement by statement to the last statement of method main()
 - Flow of control could be passed temporarily to other methods through invocations, but the control returned to main() after their completion
- Programs with single flows of control are known as sequential processes

```
Single-threaded Program {
    Statement 1;
    Statement 2;
    ...
    Statement k;
}
```

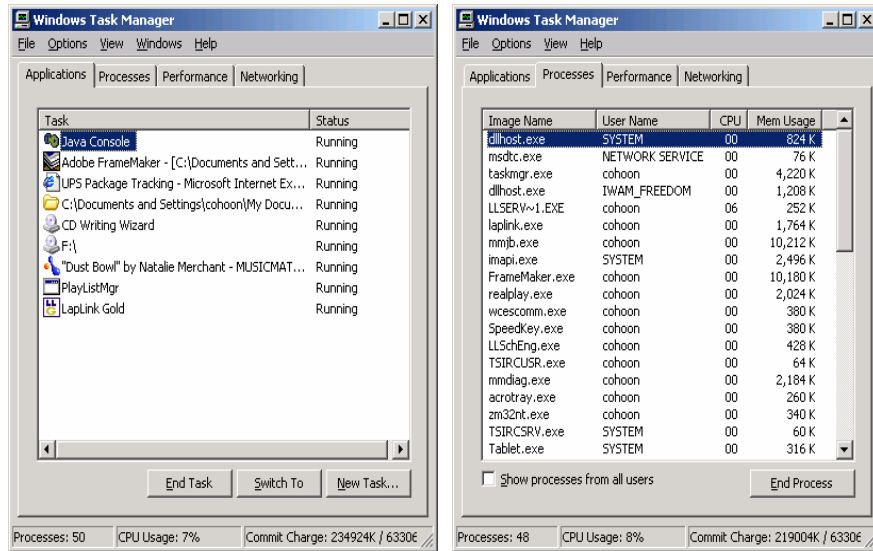
Although the statements within a single flow of control may invoke other methods, the next statement is not executed until the current statement completes



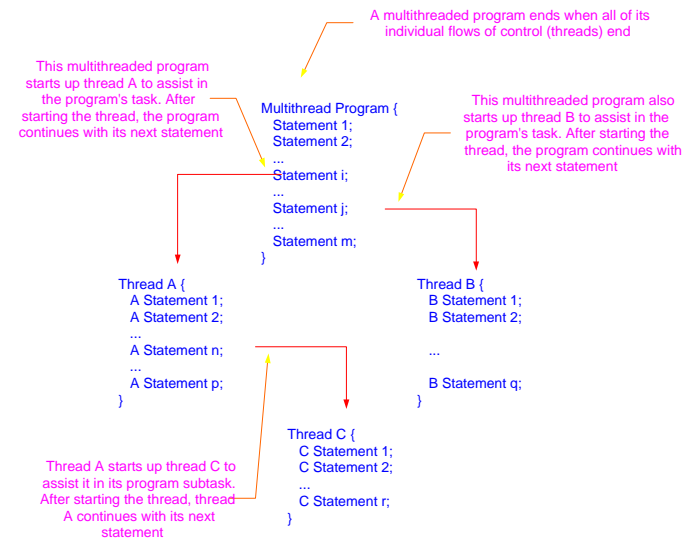
Processes

- The ability to run more than one process at the same time is an important characteristic of modern operating systems
 - A user desktop may be running a browser, programming IDE, music player, and document preparation system
- Java supports the creation of programs with concurrent flows of control – threads
 - Threads run within a program and make use of its resources in their execution
 - Lightweight processes

Processes



Multithread processing

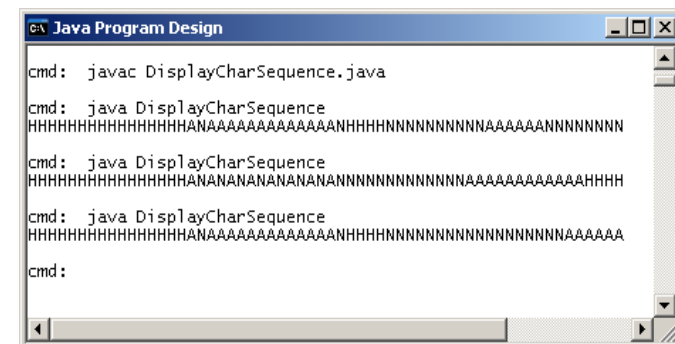


Timer and TimerTask

- Among others, Java classes `java.util.Timer` and `java.util.TimerTask` support the creation and scheduling of threads
- Abstract class `Timer` has methods for creating threads after either some specified delay or at some specific time
 - `public void schedule(TimerTask task, long m)`
 - Runs `task.run()` after waiting `m` milliseconds.
 - `public void schedule(TimerTask task, long m, long n)`
 - Runs `task.run()` after waiting `m` milliseconds. It then repeatedly reruns `task.run()` every `n` milliseconds.
 - `public void schedule(TimerTask task, Date t)`
 - Runs `task.run()` at time `t`.
- A thread can be created By extending `TimerTask` and specifying a definition for abstract method `run()`

Running after a delay

- Class `DisplayCharSequence` extends `TimerTask` to support the creation of a thread that displays 20 copies of some desired character (e.g., "H", "A", or "N")



Using DisplayCharSequence

```
public static void main(String[] args) {
    DisplayCharSequence s1 =
        new DisplayCharSequence('H');

    DisplayCharSequence s2 =
        new DisplayCharSequence('A');

    DisplayCharSequence s3 =
        new DisplayCharSequence('N');
}
```

Defining DisplayCharSequence

```
import java.util.*;

public class DisplayCharSequence extends TimerTask {
    private char displayChar;
    Timer timer;

    public DisplayCharSequence(char c) {
        displayChar = c;
        timer = new Timer();
        timer.schedule(this, 0);
    }

    public void run() {
        for (int i = 0; i < 20; ++i) {
            System.out.print(displayChar);
        }

        timer.cancel();
    }
}
```

Implementing a run() method

- A subclass implementation of `TimerTask`'s abstract method `run()` has typically two parts
 - First part defines the application-specific action the thread is to perform
 - Second part ends the thread
 - The thread is ended when the application-specific action has completed

// run(): display the occurrences of the character of interest

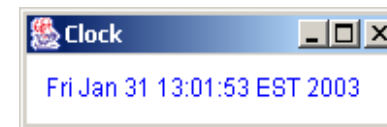
```
public void run() {
    for (int i = 0; i < 20; ++i) {
        System.out.print(displayChar);
    }
    timer.cancel();
}
```

} Desired action to be performed by thread

} Desired action is completed so thread is canceled

Running repeatedly

- Example
 - Having a clock face update every second



```
public static void main(String[] args) {
    SimpleClock clock = new SimpleClock();
}
```

```
public class SimpleClock extends TimerTask {
    final static long MILLISECONDS_PER_SECOND = 1000;
    private JFrame window = new JFrame("Clock");
    private Timer timer = new Timer();
    private String clockFace = "";

    public SimpleClock() {
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setSize(200, 60);
        Container c = window.getContentPane();
        c.setBackground(Color.white);
        window.setVisible(true);

        timer.schedule(this, 0, 1*MILLISECONDS_PER_SECOND);
    }

    public void run() {
        Date time = new Date();
        Graphics g = window.getContentPane().getGraphics();
        g.setColor(Color.WHITE);
        g.drawString(clockFace, 10, 20);

        clockFace = time.toString();
        g.setColor(Color.BLUE);
        g.drawString(clockFace, 10, 20);
    }
}
```

SimpleClock scheduling

```
timer.schedule(this, 0, 1*MILLISECONDS_PER_SECOND);
```

The millisecond delay
before the thread is
first scheduled

The number of
milliseconds between
runs of the thread

Case Studies: no need to turn in, just for practicing

- ❑ P.569 Case Study – Value-By-Value Extraction
- ❑ P.647 Case Study – Swimming Fish