Advanced Programming

"Exception Handling, Interfaces, Generic Collections, GUI and Event Handling, Network Programming, and JDBC "

Advanced Concepts in Programming

Shakirullah Waseeb
shakir.waseeb@gmail.com

Nangarhar University

May 10, 2018


Why so many languages?

Language evolution, innovation and development occurs for two fundamental reasons:

- To adapt to changing environments and uses
- To implement refinements and improvements in the art of programming

Java introduction

- Java is conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- Widely used programming language (handheld devices, network, computers)
- Java editions: Standard Edition (SE), Enterprise Edition (EE), Micro Edition (ME)


Java Buzzwords

- Java is:
  - Simple (inherit C and C++ syntax, adopted by C#)
  - Secure (confining java program to java execution environment)
  - Robust (auto memory management, error handling)
  - Portable (platform independent, byte code, JVM)
  - Object oriented (pure object oriented paradigm)
  - Multithreaded (do many things simultaneously)
  - Distributed (client/server programming, RMI)

Review on Object Technology

- Demands for new and powerful software: where quickness, economy, and correctness remains an elusive goal
- Objects are instances (single occurrence) of *classes* which are essentially *reusable software components*

Examples

Date object, time object, audio object, video object, people object etc.

- Almost any noun can be reasonably represented as a software object in terms of attributes (e.g., name, color and size) and behaviors (e.g., calculating, moving and communicating).


Review on Object Technology – Continue

Bank account example

Account Class
Attributes : account_balance, date_opened, account_type
Functions : *inquireBalance, depositAmount, withdrawAmount*

- Instantiation: The process of creating objects of a class, object being created is refered to as instance of that class.
- Reusability: Reuse of existing classes when building new classes and programs save time and efforts, also helps in building reliable and effective systems; because they passed extensive *testing, debugging and performance* tuning


Review on Object Technology – Continue

- Messages and Methods calls: Sending message to an object; message is implemented as *method call*
- Encapsulation: wrapping of attributes and methods into objects
- Inheritance: creating new classes quickly and conveniently by *absorbing* the characteristics of an existing class, possibly *customizing* them and *adding* unique characteristics of its own


Review on Object Technology – Continue

- Creating best solution requires:
  - detailed *analysis* in order to
  - determine project's *requirements* (i.e defining *what* the system is supposed to do)
  - and developing a *design* (i.e deciding *how* the system should do it) that satisfies them
- Object-Oriented Analysis and Design (OOAD): analyzing and designing system from object-oriented point of view
- Single graphical language is used for communicating the results of any OOAD process, known as *Unified Modeling Language* (UML)
- UML: graphical language for modeling object-oriented systems

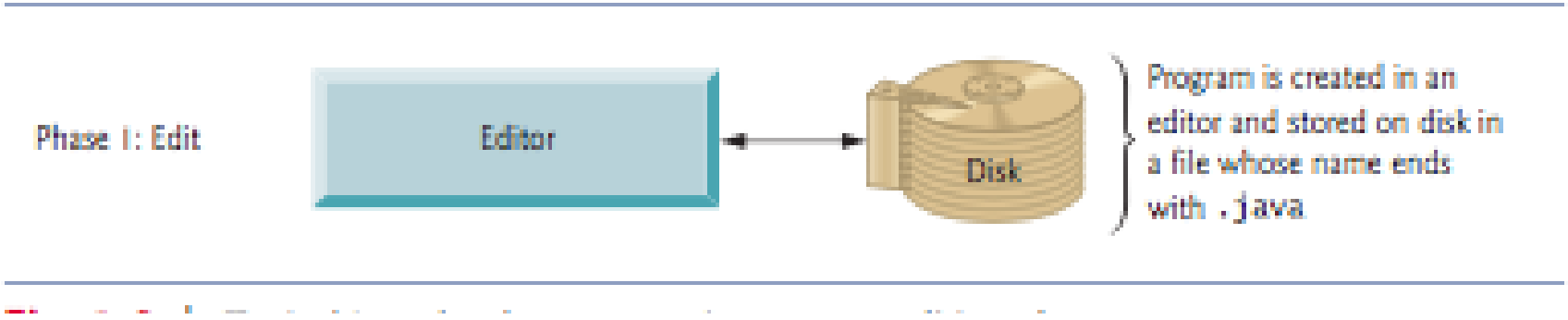

Creating and Executing Java application

- Java program creation and execution normally go through five phases – edit, compile, load, verify, and execute
- We discuss these phases in the context of the Java SE Development Kit (JDK) [1]

[1] www.oracle.com/technetwork/java/javase/downloads/index.html

Creating and Executing Java application - Phase 1

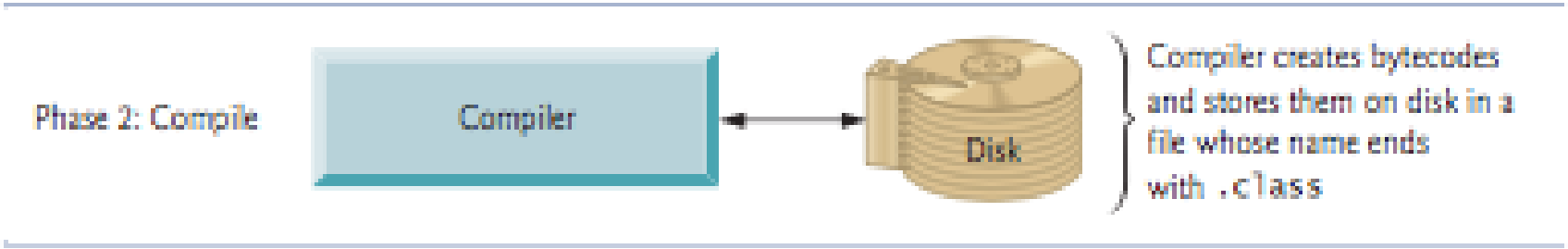
- Use an editor (vi, emacs on linux, notepad in windows)
- Type your java program typically referred as source code
- Save file with .java extension



[2]

Creating and Executing Java application - Phase 2

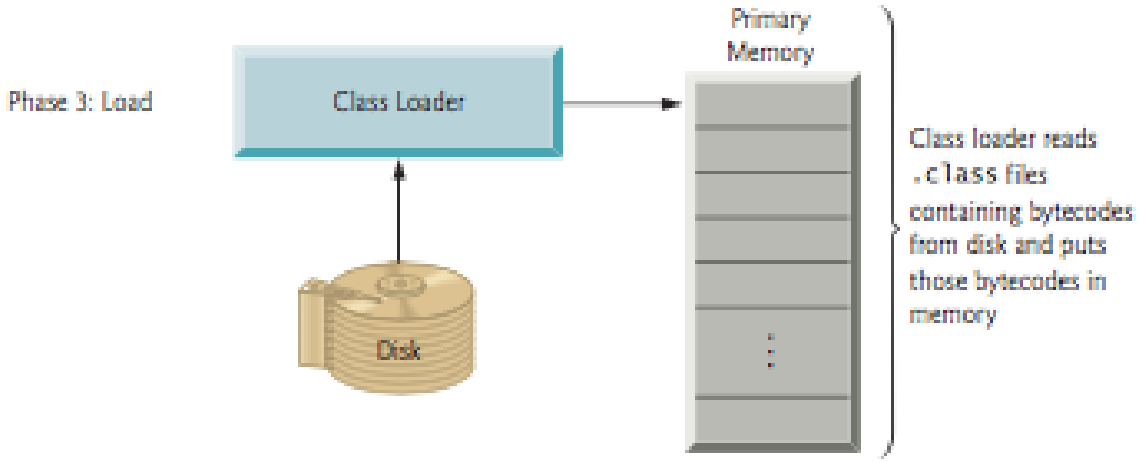
- Use command javac (java compiler) to compile the source program
- For example a program called Sallam.java we use following command javac Sallam.java
- If program compiles successfully it will produce Sallam.class file which is the compiled version of program and called bytecode



[2]

Loading a Program into Memory - Phase 3

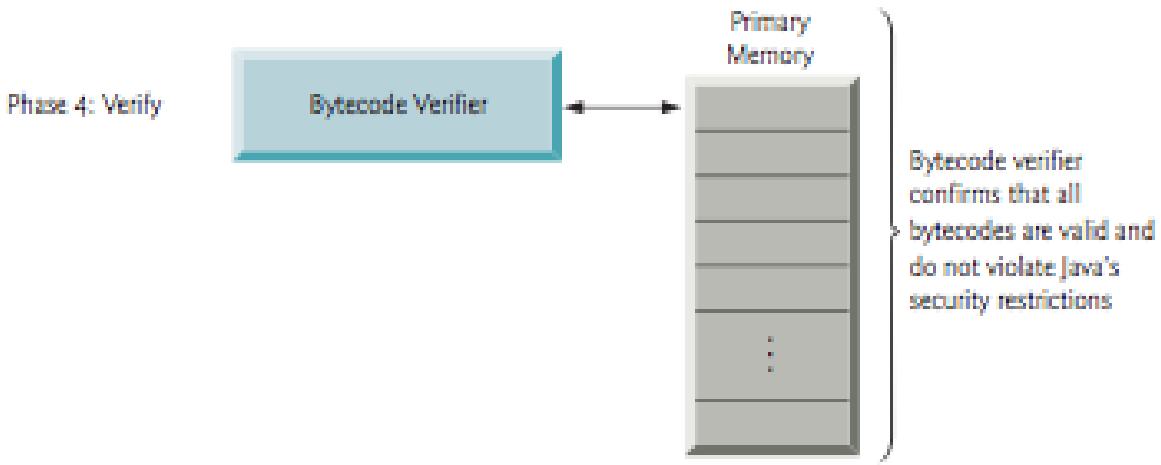
- JVM places program in memory to execute it – known as loading
- JVM's class loader takes the .class files into memory (also .class files that the program uses)
- These .class files can be loaded from hard disk or from network



[2]

Bytecode Verification - Phase 4

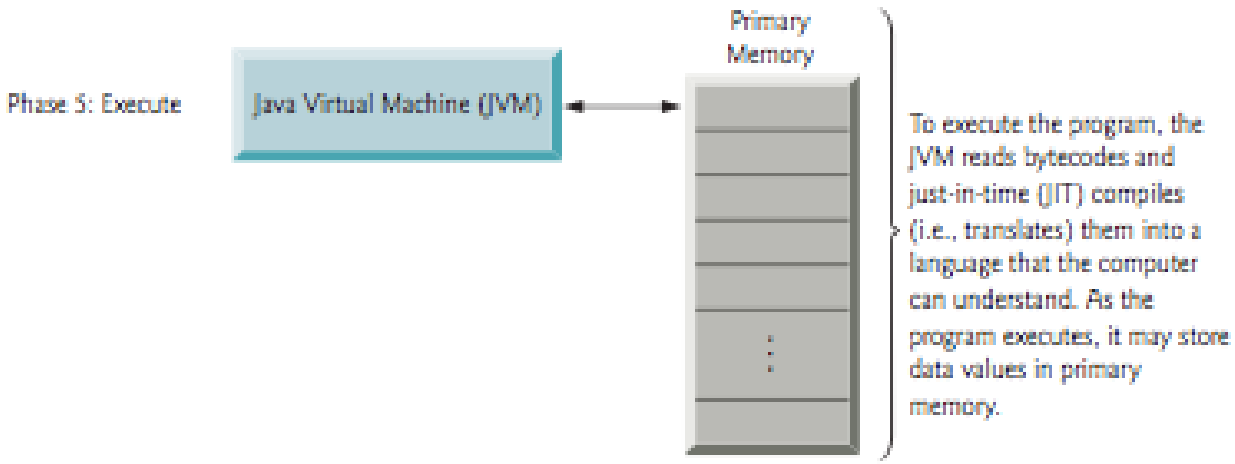
- Bytecode Verifier examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions
- Java enforces strong security



[2]

Execution - Phase 5

- JVM executes the bytecode
- Performing specified actions in the program
- These .class files can be loaded from hard disk or from network



[2]

Simple Java Program

A Simple Program Example

```
class MyClass {
        public static void main (String args[]){
            System.out.println("Assalam-o-Alikum");
            }
        }
```

- Save it as MyClass.java
- Compile it as: javac MyClass.java
- Execute it as: java MyClass


Simple Java Program for integer input

A simple example program; adding two numbers

```
class Adder {
        public static void main (String args[]){
            short num1;
            short num2;
            int sum;
            Scanner scan = new Scanner(System.in);
            System.out.println("Enter first number: ");
            num1= scan.nextShort();
            System.out.println("Enter second number: ");
            num2= scan.nextShort();
            sum= num1 + num2;
            System.out.printf(" %d + %d = %d", num1, num2, sum);
            }
        }
```


Data Types

- Java is strongly typed language
  Note: every variable has a type, every expression has a type, and every type is strictly defined
  All assignments, whether explicit or via parameters passing in methods call are checked for type compatibility
- There are no automatic coercions or conversions of conflicting types as in some languages.
- Simple Types
  - Integers: **byte, short, int, long**
  - Floating point: **float, double**
  - Characters: **char**
  - Boolean: **boolean**


Variables

- Variables are defined via a combination of type, identifier, and an optional initializer
  type identifier [= value] [, identifier [= value] ...] ;
- Dynamic initialization
- Scope and life time of variables; two general catagories global and local, however java define *class* and *method* scope
- Type conversion and type casting
  - Automatic conversion: takes place if; two types are compatible, destination type is larger than source type
  - Casting incompatible type: a cast is used to make a conversion between incompatible types: *narrowing conversion* (e.g casting a large value type into small value type ***int*** *to* ***byte***) *truncation* (e.g converting float type into integer type)
    (target-type) value
- Automatic type promotion and promotion rules


Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- One dimensional array: The general form of a one dimensional array declaration is
  type array-var[ ]; no memory will set aside
  array-var = new type[size]; memory of given size will be reserved
  array-var[0] = value1;
  array-var[1] = value2;
  array-var[ ] = {value1, value2};


Arrays –continue

- Two dimensional array: The general form of declaration is
  type array-var[ ][ ];
  array-var = new type[row-size][column-size];
  array-var[0][0] = value1;
  array-var[0][1] = value2;
  array-var[1][0] = value3;
  array-var[1][1] = value4;
  array-var[ ][ ] = {
  {value1, value2},
  {value3, value4}
  };


Arithmetic Operators

- Operators used for arithmetic calculations

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	f + 7
Subtraction	-	$p - c$	p - c
Multiplication	*	bm	b * m
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	x / y
Remainder	%	$r \bmod s$	r % s

[1]

Figure: arithmetic operators


Arithmetic Operators Precedence

- Precedence of arithmetic operators

Operator(s)	Operation(s)	Order of evaluation (precedence)
*	Multiplication	Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
-	Subtraction	
=	Assignment	Evaluated last.

[1]

Figure: precedence of arithmetic operators


Equality and relational operators

- condition is an expression that can be true or false
- e.g conditional expression in *if* selection statement, which make decision on condition's value
- Conditions in *if* statements can be formed using *equality* ($==, !=$) or *relational* ($>, <, >=, <=$) operators

Standard algebraic equality or relational operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

[1]

Figure: precedence of arithmetic operators

Class Declaration and Definition

- classes declared with *public* keyword must be:
  saved in a separate file
  file name must be same with class name

A simple class declaration example

```
public class SimpleClass {
                public void dispMessage(String str){
                      System.out.println(str);
                      }
                }
```

Save it as SimpleClass.java and compile it as javac SimpleClass.java


Instantiation and Execution

- Every java application has a class that contain a *main* method, where application starts its execution
- Some programmers refer to such class as a driver class
- Example program containing *main* method

Example program

```
public class SimpleClassApp {
        public static void main (String args[]){
            SimpleClass sc = new SimpleClass();
            sc.dispMessage("A message from application");
           }
        }
```

Save it as SimpleClassApp.java, compile it using command javac SimpleClassApp.java, and finally run it as java SimpleClassApp.


Sequential, Selectional, and Repetition structures

- Control structure:
  sequential execution; execute in the order in which program is written
  transfer of control; specify which instruction to execute next
- Sequence Structure
  - normal execution of program instruction in the order they are written
- Selection Structure
  - single selection statements (*if* statement)
  - double selection statements (*if .. else* statement)
  - multiple selection statements (*switch* statement)
- Repetition Structure
  - also called looping statements
  - looping continuation condition
  - *while*, *do while*, and *for*


Break and Continue statements

- *break* statement:
  when executed in a *while, for, do...while or switch*, causes immediate exit from that statement
  typically use to escape early from a loop or to skip the remainder of a *switch*
- *continue* statement:
  when executed in a *while, for or do...while*, skips the remaining statements in the loop body and proceeds with the *next iteration* of the loop
  while and do...while: immediately test loop-continuation
  for: increment expression executes, then loop-continuation is tested


Introduction

- Compartmentalization (dividing into groups and categories) of class name space
- To avoid class name collision
- Mechanisms for partitioning the class name space into more manageable chunks
- Naming and visibility control mechanism


Class packaging in Java

- Java uses ***package*** to compartmentalize class name space
- Class can be defined inside a **package** that are not accessible outside the package
- Even class members can be defined are only exposed to other members of the same package
- Allows classes to have intimate knowledge of each other, but not expose that knowledge to external world
- A java source file can contain any (or all) of the following four internal parts:
  - A single **package** statement (optional)
  - Any number of **import** statements (optional)
  - A single **public** class declaration (required)
  - Any number of classes **private** to the package (optional)


Defining a Package

- Quite easy:simply include a **package** statement as the first statement in a Java source file
- any classes declared in this file will belong to specified package
- **package** statement defines a name space in which classes are stored
- if package statement is omitted, class names are put into default package, having no name
- general form of **package** statement:

  package *pkgname*
- Java uses file system directories to store packages
- More than one file can include the same **package** statement
- packages hierarchy can be created using a period

  package *pkg1[.pkg2[.pkg3]]*

  package java.awt.image;


Importing Packages

- Use **import** statement to bring certain classes, or entire package, into visibility
- In Java **import** statements occur immediately following the **package** statement (if it exists) and before any class definition
- General form of **import** statement:
- **import** *pkg1[.pkg2[.classname—*]]*
- **import** *java.util.Date*
- **import** *java.lang.**


Visibility of class members

- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1. *Class Member Access*

[3]

Introduction

- Abstraction of a class from its implementation
- Specify what a class should do, but not how it does it
- Syntactically similar to classes, but lack instance variables, and methods are declared without definition
- Implementing class must create the complete set of methods defined by the interface
- However, each class is free to provide its own implementation


Interfaces in Java

- Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism
- Designed to support dynamic method resolution at runtime
- Disconnect the definition of a method or set of methods from the inheritance hierarchy


Defining an Interface

- Variables declared in interface are implicitly **static** and **final**, and must be **initialized**
- General form of java interface:

access **interface** *name* {
    return-type **method-name1** (*parameter-list*);
    return-type **method-name2** (*parameter-list*);
    .
    .
    return-type **method-nameN** (*parameter-list*);
    type **variable-name1** = **value1**;
    type **variable-name2** = **value2**;
    .
    .
    type **variable-nameN** = **valuen**;
}


Interface Example Code

Example

```
public interface calculator {
        int add (int x, int y);
        int sub (int x, int y);
        float dev (int x, int y);
        long mul (int x, int y);
        void display ();
    }
```


Implementing an Interface

- One or more classes can implement that interface
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface
- Methods that implement an interface must be declared public
- type signature of the implementing method must match exactly the type signature specified in the interface definition
- General form of implementation:
  access class **classname** [*extends superclass*][*implements interface* [,*interface...*]] {

```
        // class-body
    }
```


Interface Implementation Example Code

Example

```
public class ClassicCalculator implements calculator {
        int addResult=0;
        int subResult=0;
        int divResult=0;
        int mulResult=0;
        int add (int x, int y){
                addResult = x+y;
                return addResult;
            }
        int sub (int x, int y){
                subResult = x-y;
                return subResult;
            }
        float dev (int x, int y){
                divResult = x/y;
                return divResult;
            }
        long mul (int x, int y){
                mulResult = x*y;
                return mulResult;
            }
        void display (){
        System.out.println("Result of addition: "+addResult);
        System.out.println("Result of subtraction: "+subResult);
        System.out.println("Result of division: "+divResult);
        System.out.println("Result of multiplication: "+mulResult);
        }
    }
```

- To understand the real power of interface let's elaborate a practical example of Stack data structure
- Stack has two functions: **push** and **pop**
- Use in interface having above two functions
- Implement given interface for a fixed size stack
- Implement given interface for a dynamic size stack


What are exceptions?

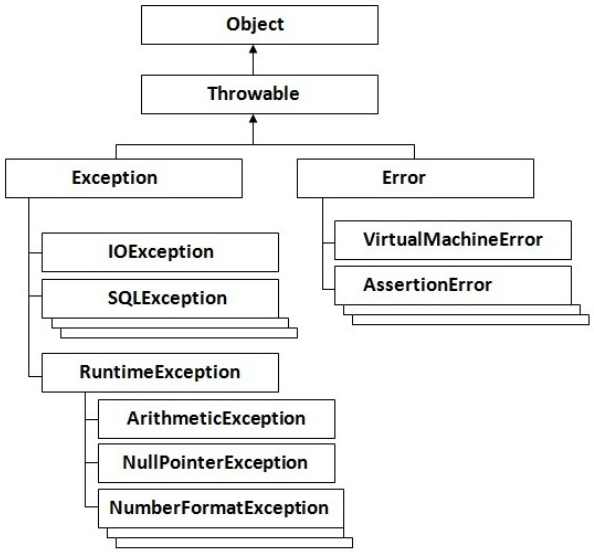
- **Exception**: errors that rise during programs execution, indication of problem that occurs during a program's execution, or an event that disrupts the normal flow of the program (e.g. divide by zero)
- **Exception Handling**: mechanism to handle such problems and errors e.g IndexOutOfBound, IO, SQL, Remote etc.
- Core advantage is to *maintain the normal flow of the application*
- There are mainly two types of exceptions:
  - Checked exceptions: exceptions that occurs at compile time, can not be ignored (e.g. IOException, SQLException etc.)
  - Unchecked exceptions: occurs at program execution time also called runtime exceptions, and are ignored at compile time. These include programming bugs, such as logic errors or improper use of an API (e.g. ArithmeticExceptions, NullPointerException etc.)
  - Errors: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer (e.g. OutOfMemoryError, VirtualMachineError etc.)


Exception Handling Terminology

- Common terms used by most of the programming languages in exception handling *throw, rise, try, except, catch, finally*
  - throw; used to trigger an exception, must have at least one catch
  - rise; same as throw statement
  - try; a block where an exception may occur and for which a particular exception will be activated
  - catch; specify the type of exception to be handled
  - except; same as catch statement
  - finally; block that run in either condition, whether an exception occur or not


Java and Exception handling

- Java Exception Handling Keywords: *try, catch, finally, throw, and throws*
- All exception classes are subtypes of java.lang.Exception class
- Exception class is a subclass of the Throwable class
- There is another subclass called Error which is derived from the Throwable class



[1]

Figure: Hierarchy of Java Exception classes

Common Java exceptions scenario

- Common scenario in which exceptions may occur:
  1. ArithmeticException

```
int a=50/0;//ArithmeticException
```

[1]

  2. NullPointerException

```
String s=null;
System.out.println(s.length());//NullPointerException
```

[1]

  3. NumberFormatException

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

[1]

  4. ArrayIndexOutOfBoundsException

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

[1]


Catching Exception

- A method catches exception using a combination of *try* and *catch* blocks.
- *try/catch* block is placed around the code which might rise an exception
- Code within *try/catch* block is referred to as protected code

Syntax [2]

```
try{
   // protected code
  } catch (ExceptionName excep){
        // code that execute in case an exception occur
       }
```


Catching Exception Example

Handling IndexOutOfBoundsException Example [2]

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest {
      public static void main(String args[]) {
            try {
                  int var[] = new int[2];
                  System.out.println("Access element three :" + var[3]);
            }catch(ArrayIndexOutOfBoundsException e) {
                  System.out.println("Exception thrown :" + e);
                  }
            System.out.println("Out of the block");
      }
}
```


throw and throws keywords

- *throws*: used to postpone the handling of a checked exception
- *throw*: used to invoke and exception explicitly

Example: throws RemoteException, InsufficientFundsException [2]

```
import java.io.*;
public class Account {
    public void deposit(double amount) throws RemoteException {
        // Method implementation
        throw new RemoteException();
    }
    public void withdraw(double amount) throws RemoteException, InsufficientFundsException {
        // Method implementation
        throw new RemoteException();
        if (this. amount < amount){
            throw new InsufficientFundsException();
        }
    }
    // Remaining of class definition
}
```


User defined exception

User defined InsufficientFundsException Example [2]

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception {
      private double amount;
      public InsufficientFundsException(double amount) {
            this.amount = amount;
      }
      public double getAmount() {
            return amount;
      }
}
```


PHP: Exception without try catch

User defined InsufficientFundsException Example [3]

```
<?php
    //create function with an exception
    function checkNum($number) {
        if($number > 1) {
            throw new Exception("Value must be 1 or below");
        }
        return true;
    }
    //trigger exception
    checkNum(2);
?>
```

Output:

```
Fatal error: Uncaught exception 'Exception'
with message 'Value must be 1 or below' in C:\webfolder\test.php:6
Stack trace: #0 C:\webfolder\test.php(12):
checkNum(28) #1 {main} thrown in C:\webfolder\test.php on line 6
```

[3]


PHP: Exception with try and catch

User defined InsufficientFundsException Example [3]

```
<?php
    //create function with an exception
    function checkNum($number) {
        if($number > 1) {
            throw new Exception("Value must be 1 or below");
        }
        return true;
    }
    //trigger exception in a "try" block
    try {
        checkNum(2);
        //If the exception is thrown, this text will not be shown
        echo 'If you see this, the number is 1 or below';
    }
            //catch exception
    catch(Exception $e) {
        echo 'Message: ' .$e->getMessage();
    }
?>
```

Output:

```
Message: Value must be 1 or below
```

[3]

Python: Exception without try catch

Python uses: *try, except, and finally for exception handling*

Example test.py

```
#/usr/bin/python
    fh = open ("testfile", "r")
```

```
Traceback (most recent call last):
  File "/Users/jank/Desktop/PythonTut/test.py", line 1, in <module>
    fh = open('testfile','r')
FileNotFoundError: [Errno 2] No such file or directory: 'testfile'
```


Python: Exception with try catch

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

[1]


What are collections?

- **Collection**: is a data structure (an object), that can hold references to other objects
- Java collections framework provides many prebuilt generic data structues
- Collections-framework interfaces [see Figure 5] declare a number of operations (e.g add, compare, remove, get, sort etc.) to be performed generically on various types of collections
- Several implementations of these interfaces are also available in the framework (e.g ArrayList, LinkedList, HashMap, PriorityQueue etc.)

Interface	Description
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does *not* contain duplicates.
`List`	An ordered collection that *can* contain duplicate elements.
`Map`	A collection that associates keys to values and *cannot* contain duplicate keys. `Map` does not derive from `Collection`.
`Queue`	Typically a *first-in, first-out* collection that models a *waiting line*; other orders can be specified.

[1]

Figure: Various Interfaces of collection-framework

Which collection to use?

- Choosing collection defends on:
  - required memory
  - methods' performance characteristics for operations such as adding, removing, searching, sorting and many more
- Review documentation of each category of collection before choosing them (e.g List, Set, Queue, and Map etc.)
- Choose appropriate implementation regarding your application's requirements (e.g LinkedList, ArrayList, HashMap, PriorityQueue etc.)


Type-Wrapper-Classes

- Java provides type-wrapper-classes corresponding each primitive type as:
  - Integer for int
  - Boolean for bool
  - Short for short
  - Character for char
  - Double for double
  - etc.
- By having these classes we can manipulate primitive types as objects
- This is helpful because data-structures such are collections can't operate on primitive data type
- All numeric type-wrapper classes extends class Number


Primitive types to type-wrapper objects

- Java provides boxing and unboxing which facilitates automatic conversion between primitive-type values and type-wrapper objects
- Boxing conversion : converts a value of a primitive type to an object of the corresponding type-wrapper class
- Unboxing conversion : converts an object of a type-wrapper class to a value of the corresponding primitive type
- These conversions are performed automatically—called autoboxing and auto-unboxing

Auto-conversion example code snippet

```
Integer[] intArray = new Integer[3]; // create an array of Integer type-wrapper
int x = 5; // create a primitive-data type variable
intArray[0] = x; // auto-boxing; assigning a primitive type value to Integer type-wrapper
x = intArray[0] ; // auto-unboxing; assigning an Integer type-wrapper to primitive type value
```


Collection interface and Collections class

- **Collection Interface**: contains bulk operations that can be performed on entire collection, such as *adding, clearing*, and *comparing* objects in a collection
  Can also be converted into array
  In addition, provides a method that return an Iterator object that walk through the collection objects and can remove objects from collection during iteration
- **Collections Class** : provides static methods that *search, sort, shuffle, copy, fill, min, max* and other operations on collections
  Also provides wrapper methods that enables to user collection as a *synchronized collection*
  Synchronized collections are for use with multithreading, which enables programs to perform operations in parallel

Employee Class

Example code for class Employee

```
// File Name : Employee.java
public class Employee{
    static int total=0;
    int id;
    String name;
    double salary;
    public Employee(String empName, double empSalary){
        total++;
        id = total;
        name = empName;
        salary = empSalary;
    }
    public void display(){
        System.out.printf("Employee name: %s \t salary: %f \n",name, salary);
    }
}
```

ArrayList Example: EmployeeDemo Class

Example code for class EmployeeDemo

```
// File Name : EmployeeDemo.java
import java.util.*;
public class EmployeeDemo{
    public static void main(String arg[]){
        ArrayList<Employee> employees = new ArrayList<Employee>();
        Scanner scaner = new Scanner(System.in);
        for(int x=0; x<3; x++){
            System.out.println("Enter employee name for :"+(x+1));
            String empName = scaner.nextLine();
            System.out.printf("Enter salary for employee %d :",(x+1));
            double empSalary = scaner.nextDouble();
            //Employee emp = new Employee(empName,empSalary);
            //employees.add(emp);
            employees.add(new Employee(empName,empSalary));
        }
        for(Employee emp: employees){
            emp.display();
        }
    }
}
```


Understanding Concurrency and Parallelism

- **Concurrency**: tasks that all are making *progress at once* (by rapidly switching between them)
- Two tasks operating in parallel, means they are executing simultaneously
- In this sense, parallelism is a subset of concurrency
- Concurrent programming has several usages consider the scenario of online audio or video streaming
  - One thread to download the stream
  - Another thread to play the downloaded stream
  - These activities proceed concurrently
- Two distinct types of multitasking are: *process-based* and *thread-based*

Introduction

- Multithreading enables us to write efficient programs that utilize the CPU usage, by keeping idle time at minimum
- Java uses threads to enable the entire environment to be asynchronous, hence prevent the CPU's waste cycles
- Single-thread systems used approach called *event loop* with *polling*
- Java eliminates this main *loop/polling* mechanisms by multithreading; one thread can pause without stoping other parts of the program


States and Life cycle of threads

- At any time a thread is said to be in one of several states as shown in Figure 7 (UML state diagram)

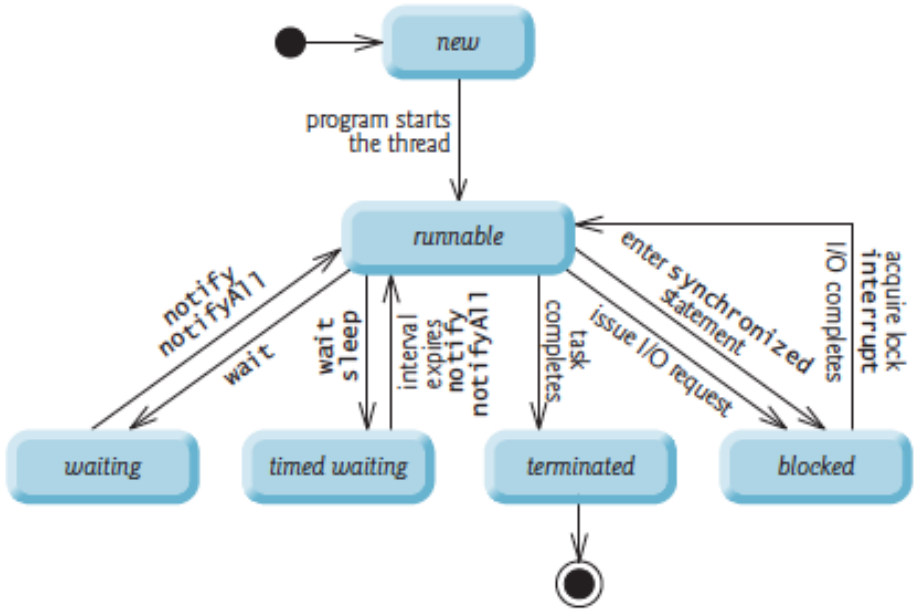


Figure: Thread life cycle [1, Page 960]

Threads priorities

- To each thread a priority is assigned, determining how this thread execution should be treated with respect to others
- Priorities are integers specifying the relative priority of one thread to another
- Thread's priority is used to decide when to switch from one running thread to the next, called *context switch*
  - A thread can voluntarily release control
  - A thread can be preempted by a higher-priority thread


Synchronization

- As multithreading introduces asynchronous behavior to program, there are situation where synchronicity is required
- In situation where threads share a common resource and perform operations on given source simultaneously
- This is required to avoid conflicts, e.g. one thread should not be allowed to write data while another thread is in the middle of reading it
- Control mechanism called *monitor* is used to model interprocess synchronization
- In Java each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object


Threads inter-communication

- Java provides easy way for two or more threads to communicate
- This can be done via calls to predefined methods that all objects have
- Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out


Thread class and the Runnable interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable
- 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 listed below

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.

Figure: Thread life cycle [3, Page 277]


The main thread

- The main thread starts running immediately when a java program starts up
- Important for two reasons
  1. It is the main thread from which other threads will be spawned
  2. Often it must be the last thread to finish execution because it performs various shutdown actions
- To get reference of current executing thread we can use following static method of class Thread as:
  Thread thread = Thread.currentThread();


Getting reference of Main Thread

Accessing Main Thread

```
//
    public class MainThreadDemo{
        public static void Main(String args[]) {
            Thread thread = Thread.currentThread();
            System.out.println("Current running thread : " + thread);
            thread.setName("Demo Main Thread");
            System.out.println("After name changed : " + thread);
        }
    }
```


Graphical User Interface (GUI)

- provides convenient way for interacting with an application
- a distinctive *look and feel*
- built from *GUI components* called *controls* or *widgets*
- GUI components are object with which user interacts with via mouse, keyboard, or another form of input such are touch and voice
- Swing GUI's components and latest API for GUI, graphics and multimedia the JavaFX


Java's look and feels

- GUI's consists of
  - **look** is its visual aspect, such as color, font, and size etc
  - **feel** components we use to interact with GUI, such as button and menu
- Swing has a cross-platform lookandfeel called *Nimbus*
- To set it as default for all applications, create a file named **swing.properties** both in *JDK* and *JRE* **lib** folder and put following line in it
  swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel


Simple GUI Example

- A simple GUI-Based Input/Output via JOptionPane

- Some fundamental Swing GUI components

Component	Description
`JLabel`	Displays *uneditable text* and/or icons.
`JTextField`	Typically *receives input* from the user.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be *selected* or *not selected*.
`JComboBox`	A *drop-down list of items* from which the *user* can make a *selection*.
`JList`	A *list of items* from which the user can make a *selection* by *clicking* on *any one* of them. *Multiple* elements *can* be selected.
`JPanel`	An area in which *components* can be *placed* and *organized*.

[2]

[2] Some basic Swing GUI components [1] Page 479

GUI Overview

- GUI programming is event-driven
- Hence, event handling is at the core of successful GUI programming
- Most events to which GUI components responds are generated by the user
- Events can be passed into GUI components in a variety of ways, depending upon the actual event
- Most common handled events are generated by:
  - Mouse
  - Keyboard
  - Various controls such as button, text box, drop down, check box etc.
- Events are supported by java.awt.event package


Event Handling Approach

- Modern approach to event handling is based on **delegation event model**
- This approach defines standards and consistent mechanism to handle and process events
- In this model *source* generates an event and sends it to one or more *listeners*
  - The listener simply waits until it receives an event
  - Once received, the listener processes the event and then returns
  - Listeners must register with a source in order to receive an event notification


Events

- In **delegation-model**, an event is an object that describes a *state change* in a source
- It can be generated as a result of users interacting with the elements in a graphical user interface
- Some of the activities that cause events to be generated are
  - pressing a button
  - entering a character via the keyboard
  - selecting an item in a list
  - clicking the mouse
- Events may also occur that are not directly caused by interactions with a user interface
  - an event may be generated when a timer expires
  - a counter exceeds a value
  - a software or hardware failure occurs
  - an operation is completed


Event Sources

- A **source** is an object that generates an event
- Occurs when the internal state of that object changes in some way
- May generate more than one type of event
- A source must register listeners in order for the listeners to receive notifications about a specific type of event
- Each type of event has its own registration method
  - general form

    public void *addTypeListener*(TypeListener el)
  - registering a keyboard listener

    public void *addKeyListener*(KeyEvent el)


Event Listeners

- A **listener** is an object that is notified when an event occurs
- Has two major requirements
  - must have been registered with one or more sources to receive notifications about specific types of events
  - must implement methods to receive and process these notifications
- The methods that receive and process events are defined in a set of interfaces found in java.awt.event
- MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved


Event-Listener Interfaces

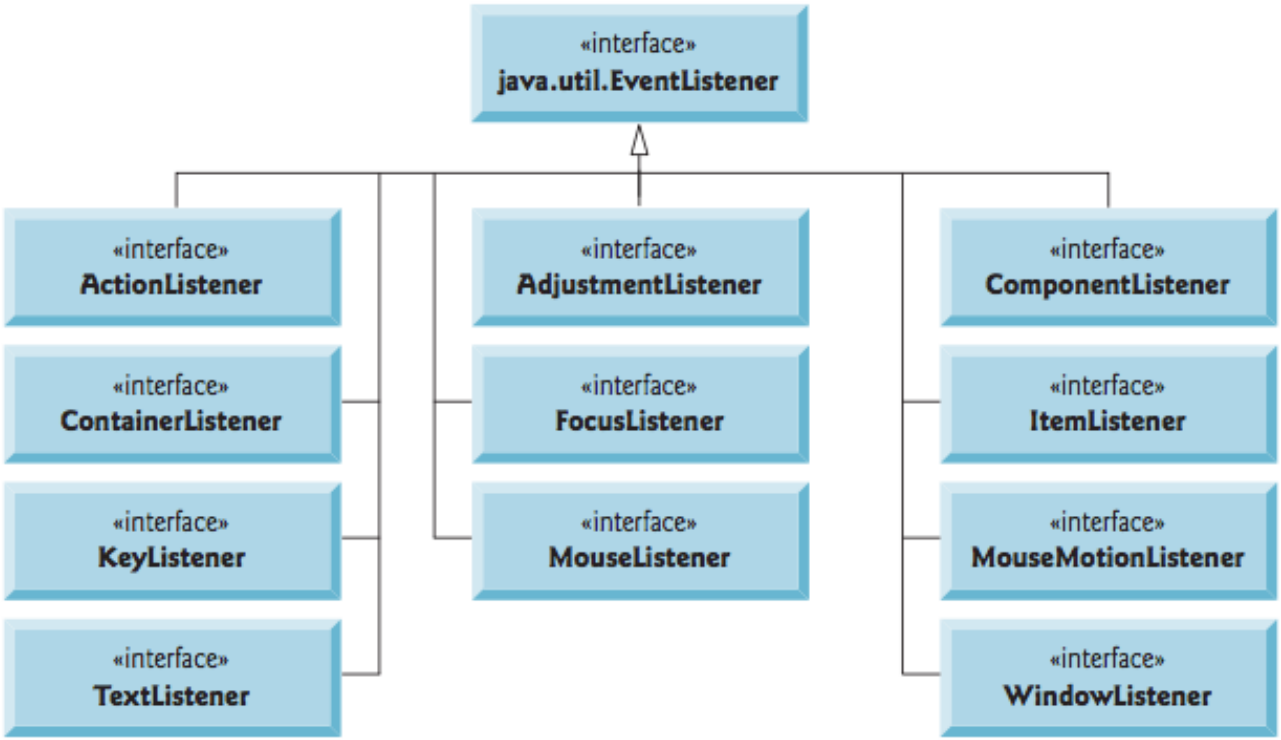


Figure: Some event-listener interfaces of package java.awt.event [1, Page 493]


Event Classes

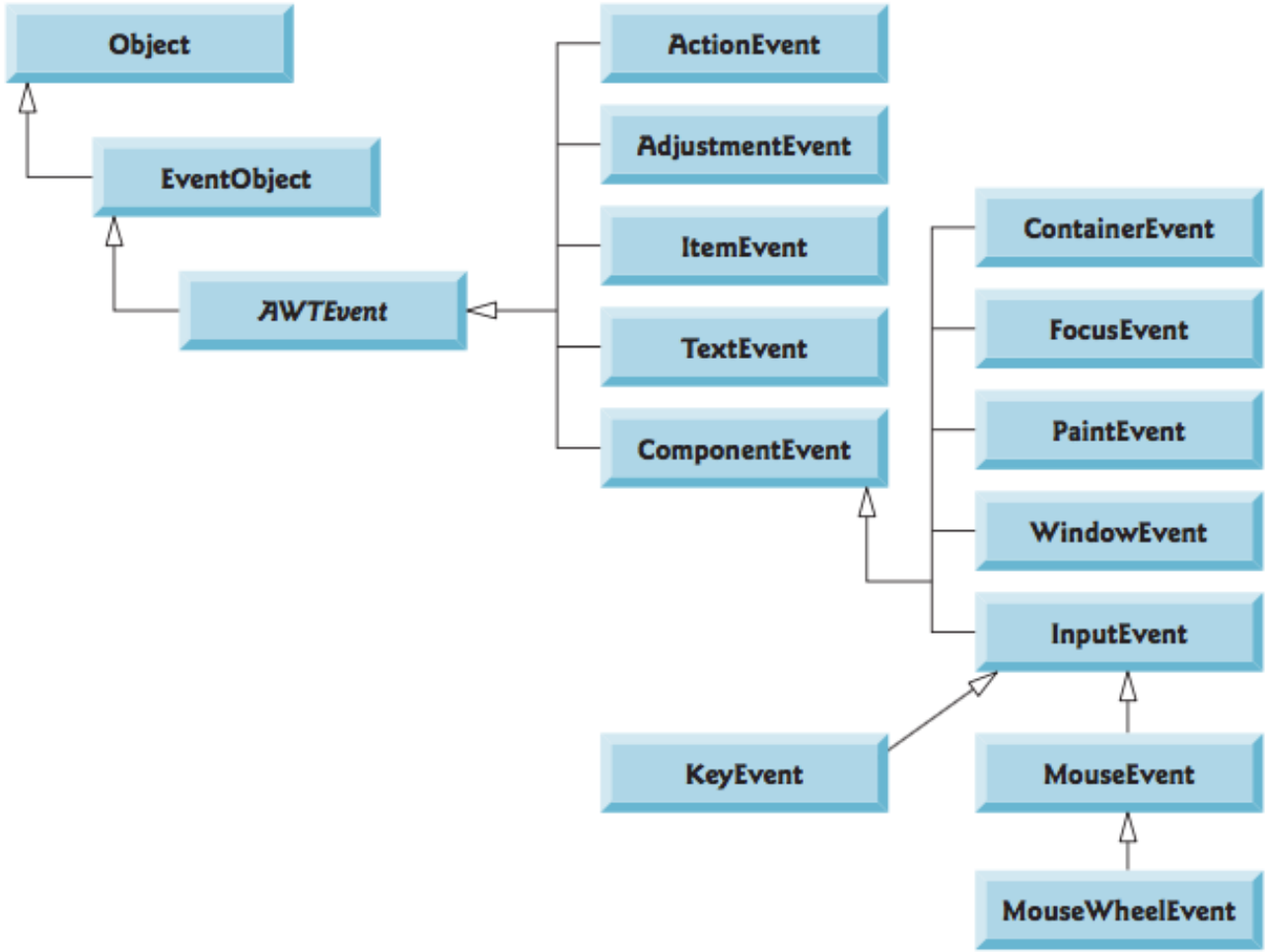


Figure: Some event classes of package java.awt.event [1, Page 492]

Event Classes

- The classes that represent events are at the core of Java's event handling mechanism
- Some of these classes are listed below

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

Figure: Some Event Classes [3, Page 657]

Example

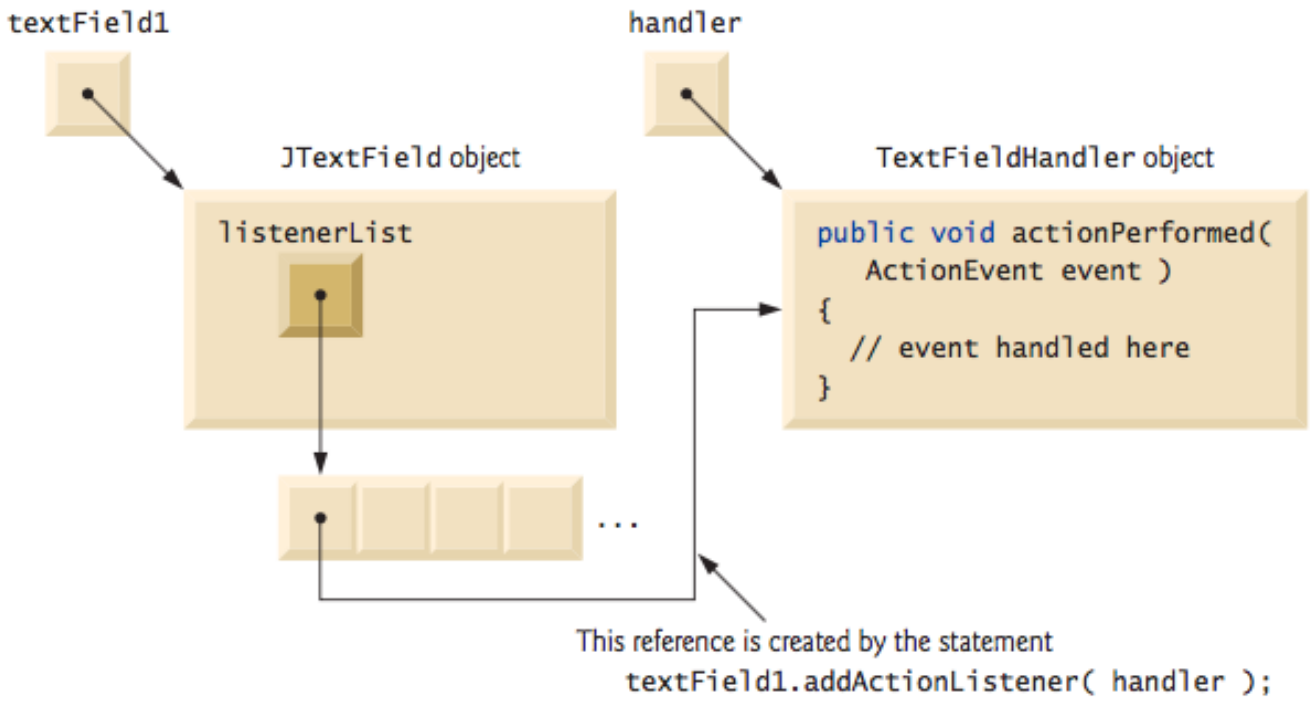


Figure: Event registration for JTestField textField1 [1, Page 494]

Introduction

- Socket programming was pioneered by Berkely Software Distribution (BSD) at University of California
- First TCP/IP communication standards
- Network Sockets
  - Similar to electrical sockets
  - Various clients can plugs around the network through sockets
  - Sockets present standard way of delivering payloads
  - IP, TCP, UDP, HTTP are some examples

Server/Client

- Server: share services to serve their clients
  - Compute server
  - Print server
  - Storage server
  - Web server
- Clients: gain access to a particular server and request for resources


Sockets Introduction

- Allows to serve many different clients at once
- Serving many different types of information
- Ports: these are numbered sockets on a particular machine
  - **Server** listening to a port until clients connects
  - **Client** connects to a particular machine on a port
  - **Session** each session is kept unique
- Reserved Sockets: TCP/IP lower than 10,24 ports are reserved for specific protocols as:
  - **21** for **FTP**
  - **23** for **telnet**
  - **25** for **email**
  - **80** for **HTTP**


Java TCP sockets

- There are two kinds of TCP sockets in Java
- ServerSocket: class is designed to be a "listener," which waits for clients to connect before doing anything
  - **Requires** a port number
  - **Listen** to clients and accepts connection to given port
  - **Returns** client socket connection object through which communicate with client
- Socket: the Socket class is designed to connect to server sockets and initiate protocol exchange
  - **Requires** a port number and address of the machine listening to given port
  - **Communicate** with server through this object once connection is established


An Example

- Here's an example [Figure 12] of a client requesting a single file, /index.html, and the server replying that it has successfully found the file and is sending it to the client:

Server	Client
Listens to port 80.	Connects to port 80.
Accepts the connection.	Writes "GET /index.html HTTP/1.0\n\n".
Reads up until the second end-of-line (\n).	
Sees that GET is a known command and that HTTP/1.0 is a valid protocol version.	
Reads a local file called /index.html.	
Writes "HTTP/1.0 200 OK\n\n".	"200" means "here comes the file."
Copies the contents of the file into the socket.	Reads the contents of the file and displays it.
Hangs up.	Hangs up.

Figure: http example [3, Page 590]


Server Example Code

```
import java.net.*;
import java.io.*;
0 references
public class Server{
    static ServerSocket listener;
    static Socket soc;
    static PrintWriter out;
    static BufferedReader in;
    0 references
    public static void main(String args[]){
        try{
                listener = new ServerSocket(9060);
                soc = listener.accept();
                out = new PrintWriter(soc.getOutputStream(),true);
                out.println("This is server khan");
        } catch (IOException ex){
            System.out.println(ex);
        } finally{
             try{
               soc.close();
                } catch (IOException ex){
                        System.out.println(ex);
                        }
                }
        }
    }
```

Figure: Server Example Code

Client Example Code

```
import java.net.*;
import java.io.*;
0 references
public class Client{
    static Socket soc;
    static BufferedReader in;
    0 references
    public static void main(String args[]){
        try{
                soc = new Socket(args[0],9060);
                in = new BufferedReader(new InputStreamReader(soc.getInputStream()));
                System.out.println("Message from server " + in.readLine());
                }catch(IOException ex){
                        System.out.println(ex);
                }finally{
                        try{
                                soc.close();
                                }catch(IOException ex){
                                        System.out.println(ex);
                                }
                        }
                }
        }
```

Figure: Client Example Code


References

P.J. Deitel, H.M. Deitel
Java How to program, 10th Edition .
Prentice Hall, 2015.

P.J. Deitel, H.M. Deitel
Java How to program, 9th Edition .
Prentice Hall, 2012.

Herbert Schildt
The complete reference Java2, 5th Edition .
McGraw-Hill/Osborne, 2002.


References

Java Point

Exception Handling in Java

http://www.javatpoint.com/exception-handling-in-java , date visited Feb 18, 2017.

Tutorials Point

Java - Exceptions

https://www.tutorialspoint.com/java/java_exceptions.htm , date visited Feb 18, 2017.

W3 Schools

PHP Exception Handling

https://www.w3schools.com/php/php_exception.asp , date visited Feb 18, 2017.

References

Tutorials Point
Python Exceptions Handling
https://www.tutorialspoint.com/python/python_exceptions.htm , date visited Feb 18, 2017.