Java Programming

Amit Gulati
amit.gulati@gmail.com

1

Java Technology

Amit Gulati
amit.gulati@gmail.com

2

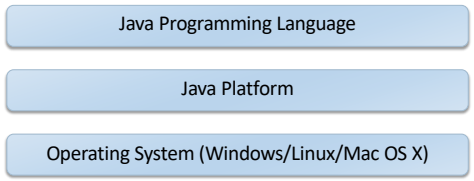
Java Technology

- Java has been used in a variety of environments and for variety of purposes
  - Desktop
    - Applications
    - Java Applets to be executed in Browser environment.
  - Web
    - J2EE Applications
    - JSP and Servelets
  - Mobility
    - J2ME
    - Blackberry
    - Android Application Development

3

Java Technology

- Comprised of
  - Java Programming Language
  - Java Platform
    - Java Virtual Machine (JVM)
    - Java API



4

Java Programming Language

- 'C' Style syntax
  - Syntax of Java is similar to other C style programming languages like C/C++/C# etc.
  - If you know C++ , you will feel at home with Java.
  - Java Source

```
package org.training;

public class HelloJava {

    //constructor
    public HelloJava(){

    }

    public static void main(String[] args) {
        System.out.println("Hello Java");
    }
}
```

5

5

Java Programming Language

- Object Oriented Language
  - Classes and Objects
  - No support for Functional Programming model
    - You have to start with at-least one class for a program.

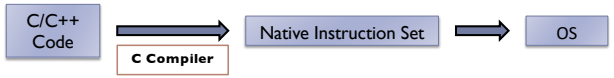
```
package org.training;

public class HelloJava {

    //constructor
    public HelloJava(){

    }

    public static void main(String[] args) {
        System.out.println("Hello Java");
    }
}
```

6

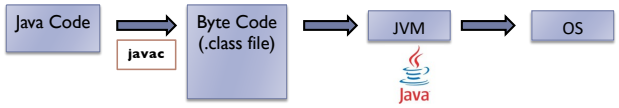
6

Java Programming Language

- Java is an Interpreted Language
  - C/C++ are native languages.



  - Java Source gets converted to byte code that is interpreted by Java Runtime.



Java Programming Language

- Write once and Run Anywhere
  - Java source code is not compiled into native code (CPU instructions).
  - Java source is converted into Java Byte Code.

```java
public class HelloWorld {

    public static void main(String[] srgs)
    {
        int i = 100;

        System.out.println("Hello World " + Integer.toString(i));
    }
}
```

.java

```
//    0    0:bipush          100
//    1    2:istore_1

//    2    3:getstatic       #16  <Field PrintStream System.out>
//    3    6:new             #22  <Class StringBuilder>
//    4    9:dup
//    5   10:ldc1            #24  <String "Hello World ">
//    6   12:invokespecial   #26  <Method void StringBuilder(String)>
//    7   15:iload_1
//    8   16:invokestatic    #29  <Method String Integer.toString(int)>
//    9   19:invokevirtual   #35  <Method StringBuilder StringBuilder.append(String)>
//   10   22:invokevirtual   #39  <Method String StringBuilder.toString()>
//   11   25:invokevirtual   #42  <Method void PrintStream.println(String)>
//   12   28:return
```

.class

Java Programming Language

- Write once and Run Anywhere
  - Java Byte Code is interpreted by the Java Virtual Machine (JVM)

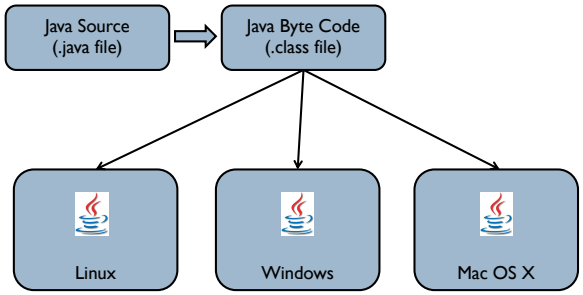
```
//    0    0:bipush          100
//    1    2:istore_1

//    2    3:getstatic       #16  <Field PrintStream System.out>
//    3    6:new             #22  <Class StringBuilder>
//    4    9:dup
//    5   10:ldc1            #24  <String "Hello World ">
//    6   12:invokespecial   #26  <Method void StringBuilder(String)>
//    7   15:iload_1
//    8   16:invokestatic    #29  <Method String Integer.toString(int)>
//    9   19:invokevirtual   #35  <Method StringBuilder StringBuilder.append(String)>
//   10   22:invokevirtual   #39  <Method String StringBuilder.toString()>
//   11   25:invokevirtual   #42  <Method void PrintStream.println(String)>
//   12   28:return
```

9

9

Java Programming Language

- Write once and Run Anywhere
  - Operating Systems for which JVM is available are able to execute Java Byte code without recompile.



10

10

Java Programming Language

- Multithreading Support
  - Multithreading support in case of C/C++
    - Win32
    - Posix
    - Pthreads library
    - High complexity
  - Java Virtual Machine provides multi-threaded execution context to applications written in Java
    - Applications use the same Java classes to add multi-threading support.
    - Implementation, debugging etc. becomes simple

11

11

Java Platform

- Java API
  - A programming language allows us to implement the logic of the program.
  - In addition to logic, we need the following for implementing applications
    - Data Structures
    - File System access
    - Networking access
    - Multithreading support
    - Database
    - Etc.
  - Example: Win32, MFC, Qt libraries

12

12

Java vs C++

C++	Java
Compiled to Native machine code (Platform Dependent)	Compiled to Byte code (Interpreted by JVM)
Write Once, compile anywhere	Write once, run anywhere
Direct access to native system	Runs in protected Virtual machine.
Explicit memory management & pointers	Managed memory access
Multiple Inheritance	Single Inheritance
Supports Operator Overloading	No Operator Overloading

15

15

Java Development Environment

Amit Gulati
amit.gulati@gmail.com

16

16

JDK & JRE

- Java JRE (Java Runtime Environment)
  - Includes tools and settings that are required to run a Java program.
    - Sun's implementation of JVM.
    - Environment Variables
- Java JDK (Java Development Kit)
  - Includes everything that is required to build application using the Java programming language.
    - Development Tools
    - Core Libraries
    - Reference Documentation
    - JRE
    - Etc.

17

17

Development Tools

- When you install JDK, development tools like compiler, runtime etc. get installed
  - javac – The Java compiler
  - java – Java Run time
  - jar – Java packger
  - javadoc – Generate document using comments added to source.
  - Jarsigner – Utility for signing jar files
  - etc….

20

20

Development Environment

- When writing Java Applications there are a few options available
  - Using an Integrated Development Environment
    - Eclipse
    - IntelliJ
    - NetBeans
    - JBuilder
  - Without an IDE
    - Source Code Editing
      - Use your favorite text editor for writing source code
      - Emacs, vi, notepad++ etc.
    - Compilation
      - Javac compiler using command line (terminal)
      - Ant Build scripts

21

21

Data Types & Named Values

Amit Gulati
amit.gulati@gmail.com

25

Basics

- Data Types
  - Java is a strongly typed language.
  - Every variable must have an associated type.
  - Data Types can be categorized as
    - Primitive / Built-in Types
    - Pre-defined Object Types
    - User Defined Types

26

Data Types

- Java Primitive Types

Primitive Type	Reserved Word	Size	Min Value	Max Value
Boolean	`boolean`	--	--	--
Character	`char`	16-bit	Unicode 0	Unicode $2^{16} - 1$
Byte integer	`byte`	8-bit	-128	+127
Short integer	`short`	16-bit	-2^{15}	$+2^{15} - 1$
Integer	`int`	32–bit	-2^{31}	$+2^{31} - 1$
Long integer	`long`	64-bit	-2^{63}	$+2^{63} - 1$
Floating-point	`float`	32–bit	IEEE 754	IEEE 754
Double precision floating-point	`double`	64-bit	IEEE 754	IEEE 754

27

Data Types

- Java Primitive Types
  - Allocated on the stack
  - Exhibit copy semantic

Named Values

- Variables
  - Used to store values in a program.
  - Declaring Variables

```
float speed;
float acceleration;
int itemCount;
```

  - What does declaration of variables do?
    - Allocation of memory to store the value of variable.
    - Number of bytes allocated depends on the type of variable

Named Values

- Initializing Variables
  - Assigning value to variable

```
speed = 100.0f;
acceleration = 2.0f;
itemCount = 50;
```

  - As a general rule, variable should not be used unless it has been initialized.
  - Declaring and initializing in one step

```
float speed = 100.0f;
float acceleration = 2.0f;
int itemCount = 50;
```


Named Values

- Constants
  - Variables whose value does not change during the execution of a program are known as constants.
  - In Java constants are marked using "**final**" keyword.
  - Format for defining a constant

```
final datatype CONSTANTNAME = VALUE;
```

  - Example

```
final double PI = 3.14159;
```

  - A variable marked as "final" cannot be re-assigned.
    - It is initialized only once.

Basics

- Naming Convention
  - Java uses "Camel Case" naming convention.
    - Each words initial letter is capitalized

```
thisIsMyMethod
```

  - Method names begin with lower case.

```
public void thisIsMyMethod()
```

  - Class names begin with upper case.

```
public class MyClass
```

  - Constants are all upper case

```
public final static int NUMBER_OF_OBJECTS
```


34

Operators

- Arithmetic Operators

+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

35

Operators

- Unary Operators

+	Unary plus operator; indicates positive value (numbers are positive without this, however
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

36

Operators

- Equality, Relational Operators

==	> greater than >= greater than or equal to < less than <= less than or equal to
!=	Not Equal to
>	Greater than
>=	Greater than equal to
<	Less than
<=	Less than equal to

37

Operators

- Conditional Operators

Conditional	?:	Given *operand1* ? *operand2* : *operand3*, where *operand1* must be of Boolean type, return *operand2* if *operand1* is true or *operand3* if *operand1* is false. The types of *operand2* and *operand3* must agree.	1
Conditional AND	&&	Given *operand1* && *operand2*, where each operand must be of Boolean type, return true if both operands are true. Otherwise, return false. If *operand1* is false, *operand2* is not examined. This is known as *short-circuiting*.	3
Conditional OR	||	Given *operand1* || *operand2*, where each operand must be of Boolean type, return true if at least one operand is true. Otherwise, return false. If *operand1* is true, *operand2* is not examined. This is known as *short-circuiting*.	2

Data Types

- Type Casting
  - Implicit

    ```
    double var = 10;
    ```

  - Explicit

    ```
    int var = (int) 10.0f;
    ```

  - Truncation
    - Type casting can lead to subtle logic errors in your program.
    - If type casting from a larger type variable (double) to a smaller type (int), value is truncated.

You cannot cast between boolean values and any numeric type.

Comments

- Java Comments
  - Being a C style programming language, Java supports two styles of comments:
    - Anything following two forward slashes (//) and before the end of the line is a comment.
      Example:

      // This is a comment

    - Anything enclosed between /* and */ is also a comment:
      Example:

      /* This is the other style of comment */

As a general coding practice,
single line comments use the first form i.e. //, and
multiple line comments use the later format i.e. /* */

41

Strings

- String Type
  - sequences of Unicode characters

```
String e = ""; // an empty string
String greeting = "Hello";
```

  - Part of the standard Java library
  - Reference Type

```
String middleName = null;
```

  - Strings are immutable in Java

45

Strings

- String Concatenation
  - use + to join (concatenate) two strings.

```
String name = "Amit";
String greeting = "Hello";
String message = greeting + " " + name;
```

  - concatenate a string with a value that is not a string

```
int age = 13;
String rating = "PG" + age;
```


46

String

- String Equality
  - Don't use == operator, use the **equals** method.
  - Case sensitive comparison

```
String greeting = "Hello";
"Hello".equals(greeting);
```

  - Case in-sensitive comparison

```
"Hello".equalsIgnoreCase("hello");
```


47

Strings

- Building Strings
  - Using String concatenation to create a larger string from smaller strings becomes inefficient.
  - **StringBuilder** class allows us to build a string from many small pieces

```
StringBuilder value = new StringBuilder();
value.append("Amit");
value.append(" ");
value.append("Gulati");
value.toString();
```


48

Strings

- Building Strings
  - **StringBuffer** class also allows us a thread safe way to build a string from many small pieces

```
StringBuffer value = new StringBuffer();
value.append("Amit");
value.append(" ");
value.append("Gulati");
value.toString();
```


49

Object Oriented Programming

Amit Gulati
amit.gulati@gmail.com

50

Object Oriented Programming

- Object-oriented programming (OOP) is a style of programming that organizes programs into
  - Collections of objects that interact with each other.
- An Object has a state, and behavior.
  - State of the Object is represented by the instance variables defined in the class of the object.
  - Behavior of the Object is represented by the instance methods defined in the class of the object.
- Objects interact with each other by sending messages to each other.
  - Messages are nothing but method calls.

51

Object Oriented Programming

- Fundamental Concepts
  - Encapsulation
    - Also known as Information Hiding, refers to hiding the inner workings of a class from the users of a class.
    - Users of class object only know what methods are implemented by the class, rather than how they are implemented.
    - Objects can be manipulated only by a defined interface: the set of methods its class implements.

52

Object Oriented Programming

- Fundamental Concepts
  - Inheritance
    - Provides ability to re-use code
    - We can establish a parent-child relationship among the classes.
      - Child class inherits data and functionality from the parent.
    - Inheritance provides a way to create new classes by extending or modifying the behavior of an existing class.
    - Java supports single inheritance.
      - A class can inherit from a single parent.

53

Object Oriented Programming

- Fundametal Concepts
  - Polymorphism
    - Polymorphism is the capability of objects belonging to different classes to respond to the same message.
    - The caller object does not need to know what type of object is responding to the message, just that there is an object that is responding.

54

Classes

- Classes define the blue prints from which objects are constructed.
- Classes define the attributes and behaviors for a class of real objects.
- Example: Rectangle class

Rectangle Class

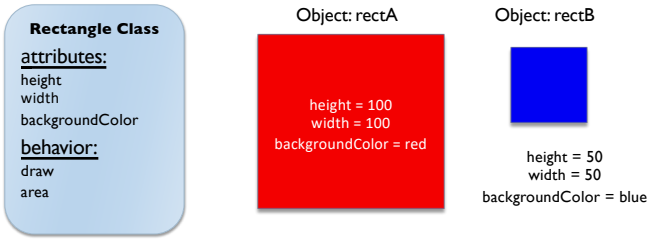
attributes:
height
width
backgroundColor

behavior:
draw
area

55

Classes

- Class names generally begin with capital letter.
- Instances of class
  - Objects created using a class are known as instances of the class.
  - **rectA** and **rectB** are instances of Rectangle class.



Classes

- Java classes
  - Unlike C++, in Java there is no separation between class declaration and definition.
  - Declaring/Defining classes in Java

```
<access specifier> class <name> extends <super class>
                                    implements <interfaces> {
        //class body
        fields, methods etc.
}
```

Optional

Java Classes

- Member Variables
  - Instance Variables
    - Define the state of an Object.
  - Declaring Member Variables
    - Instance Variables

```
<access modifier> <data type> <name of the variable>
```


58

Java Classes

- Member Variables
  - Class Variables
    - Not associated with a particular class instance.
  - Declaring Member Variables
    - Class Variables

<access modifier> **static** <data type> <name of the variable>

59

Java Classes

- Member Variables
  - Access modifiers
    - **private** : Member variable is only accessible from within the class.
    - **protected** : Member variable accessible from sub-classes.
    - **public** : Member variable accessible from outside the class and class hierarchy.

60

Java Classes

- Member Variables
  - Initializing Member Variables
    - By default, member variables are set to zero values.
    - Initialized where they are declared.

```
class WeatherData
{
    int field1 = 100;
    String country = "United States";
    String[] cities = {"Chicago", "New York", "Los Angeles"};
    double[][] temperatures = {{0.0, 0.0}, {0.0, 0.0}, {0.0, 0.0}};
}
```

    - Constructor

```
public WeatherData() {
    field1 = 100;
    country = "United States";
    cities =  new String[]{"Chicago", "New York", "Los Angeles"};
    temperatures = new double[][]{{0.0, 0.0}, {0.0, 0.0}, {0.0, 0.0}};
}
```


61

Java Classes

- Member Variables
  - Read only Instance Variable
    - "**final**" keyword is used to mark variables as read-only.
    - Each object receives its own copy of a read-only instance field.
    - A final member variable must be initialized
      - At point of declaring
      - In constructor

```
public class HelloWorld {
    final int instanceVar;
    static int classVar;

    public HelloWorld() {
        instanceVar = 100;
        classVar = 200;
    }
```


Java Classes

- Member Variables
  - Read only Class Variable
    - "**final**" keyword is used to mark class variables as read-only.
    - A final class member variable must be initialized when it is declared.
    - final class variable is a true constant.

```
public class HelloWorld {
    final int instanceVar;
    final static int classVar = 200;
    final static String strClassVar = "HelloWorld";

    public HelloWorld() {
        instanceVar = 100;
    }
```

Java Classes

- Member Methods
  - Instance Methods
    - Have direct access to Instance variables of a class.
    - Manipulate the state of the object (state defined by instance variables).
  - Declaring Member Methods
    - Instance Methods

```
<access modifier> <return type> <method name>( <parameter list> )
                                                    throws <exception list>
```

```
class Rectangle {
    int width;
    int height;

    int area() {
        return width * height;
    }
}
```


64

Java Classes

- Member Methods
  - Class Methods
    - Not related to a particular instance of a class.
    - Do not have direct access to Instance variables.
  - Declaring Member Methods
    - Class Methods

```
<access modifier> static <return type> <method name>( <param list> )
                                                    throws <exception list>
```

```
class Rectangle {
    static int NUMBER_OF_RECTANGLES = 0;
    ……
    ……
    static int getNumberOfRectangles() {
        return NUMBER_OF_RECTANGLES;
    }
}
```


65

Java Class

- Member Methods
  - Method Overloading
    - Java allows defining of methods with same name, as long as they have different set of parameters

```
public class Logger {
    public void log(String str){
        //log a string
    }
    public void log(int i) {
        //log an integer
    }

    public void log(float f) {
        //log a float
    }
}
```


66

Java Classes

- Constructors
  - Constructors are special types of methods that are used to initialize objects
    - Same name as the class name.
    - No return value.
    - Initialize the state of object.

```
public class Logger {

    byte[] buffer;

    public Logger() {
        buffer = new byte[1024];
    }
```


68

Java Classes

- Constructors
  - Default Constructor
    - Constructor that takes no parameters is known as Default Constructor.
    - If a class does not define any constructor, a default constructor is synthesized by the compiler.

```
public class Logger {

    byte[] buffer;

    public Logger() {
        buffer = new byte[1024];
    }
```


Java Classes

- Constructors
  - Parameterized Constructor
    - Constructor that takes parameters

```
public Logger(byte[] buf)
{
    buffer = buf;
}
```

Java Classes

- Constructor Overloading
  - A class can have many constructors

```
class Rectangle {
    int width;
    int height;

    Rectangle() {
        width = 0;
        height = 0;
    }

    Rectangle(int w, int h){
        width = w;
        height = h;
    }

    Rectangle(int w) {
        width = w;
        height = 0;
    }
}
```


71

Java Classes

- Constructor Overloading
  - Calling another constructor from constructor

```
class Rectangle {
    int width;
    int height;

    Rectangle(int w, int h){
        width = w;
        height = h;
    }
    Rectangle() {
        this(0, 0);
    }

    Rectangle(int w) {
        this(w, 0);
    }

}
```


72

Java Classes

- Class Initializer
  - A class initializer is a compound statement marked as "**static**", that is introduced into a class body.

```
public class JDBCFilterDriver implements Driver
{
   static private Driver d;
   static
   {
      // Attempt to load JDBC-ODBC Bridge Driver and register that
      // driver.
      try
      {
         Class c = Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         d = (Driver) c.newInstance();
         DriverManager.registerDriver(new JDBCFilterDriver());
      }
      catch (Exception e)
      {
         System.out.println(e);
      }
   }
```


Java Classes

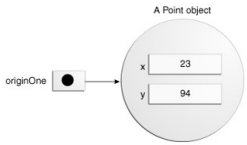
- Creating Instances
  - Java objects are allocated on the heap.
  - The only way to create Java objects is to use the "new" operator.

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }

    public static void main(String[] args)
    {
        Point originOne = new Point(23, 94);
    }
}
```

Java Classes

- Creating Instances
  - Memory Model

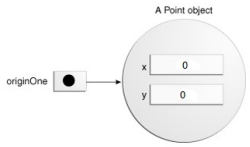
```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }

    public static void main(String[] args)
    {
        Point originOne = new Point(23, 94);
    }
}
```



Java Classes

- Passing Java Objects to methods
  - Primitive types in java are always passed as value.
  - Java Objects are always passed as reference.

```
public static void main(String[] args)
{
    Point originOne = new Point(23, 94);
    makeZero(originOne);
}

public static void makeZero(Point p)
{
    p.x = 0;
    p.y = 0;
}
```



Java Classes

- **this** reference
  - Every object can access a reference to itself with keyword this.
  - A method's body *implicitly* uses keyword `this` to refer to the object's instance variables and other methods.
  - Can use the `this` reference implicitly and explicitly

```
class Rectangle {
    int width;
    int height;

    void setWidth(int width) {
        this.width = width;
    }
}
```


79

Java Classes

- Composition
  - A class can have references to objects of other classes as members.
  - This is called composition and is sometimes referred to as a *has-a* relationship

```
class Color {
    float red, green, blue, alpha;
}

class Rectangle {
    int width;
    int height;
    Color backgroundColor;
```


80

Java Enum

- Enum
  - The basic enum type defines a set of constants represented as unique identifiers.
  - Like classes, all enum types are reference types.
  - An enum type is declared with an enum declaration, which is a comma-separated list of *enum constants*
  - The declaration may optionally include other components of traditional classes, such as constructors, fields and methods.

81

Java Memory Management

- Garbage Collection
  - Every object uses system resources, such as memory.
  - The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used.
    - When there are *no more references* to an object, the object is *eligible* to be collected.
    - Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates.

82

Java Memory Management

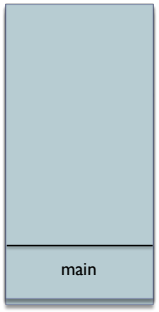
- Garbage Collection
  - Memory leaks are less likely in Java, but some can still happen in subtle ways.
  - Reference Cycles

83

Java Memory Model

- Stack based allocation
  - Creating variables on a stack.

```
public class MyClass {
public void myFunc() {
    int width;
    float height;
    double area;

    //do something
}

public static int main() {
    MyClass.myFunc();

    //call another function
}}
```



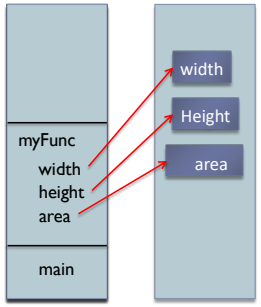
Timmins Training Consulting

84

Java Memory Model

- Heap based allocation
  - Creating variables on a stack.

```
public class MyClass
public static void myFunc() {
    Integer width = new Integer(4);
    Float height = new Float(4.0f);
    Double area = new Double(8.0);

    //do something
}

public static int main() {
    MyClass.myFunc();

    //call another function
}}
```



85

Java Memory Model

- Primitive Types memory allocation
  - All primitive types if defined at function scope are allocated on **stack**.

```
int width;
float height;
```

Built-in types

  - There is no way in java to allocate primitive types on the heap.

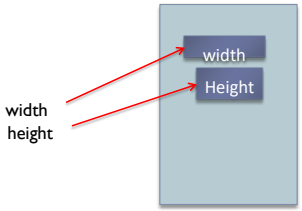
```
int width = new int(4);
float height = new float(4.0f);
```


86

Java Memory Model

- Objects types memory allocation
  - Memory for Objects comes from Heap.
  - References to Objects are stored on the stack.

```
Integer width = new Integer(4);
Float height = new Float(4.0f);
```



87

Java Memory Management

- Finalize Method
  - **finalize**() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
  - A subclass can override the **finalize**() method to dispose of system resources or to perform other cleanup.
  - The virtual machine might never call **finalize**() before an application terminates, you should provide an explicit cleanup method.
    - Use finalize as the last case scenario.
  - If you override finalize() method, call super.finalize();

88

Class Design Hints

- Always keep Data Private
  - Provide access to properties via public methods (accessors and mutators)
- Always initialize data
  - Java won't initialize local variables for you, but it will initialize instance fields of objects.
  - Don't rely on the defaults, but initialize all variables explicitly.
- Don't use too many basic types in a class.
  - The idea is to replace multiple related uses of basic types with other classes.

89

Class Design Hints

- Don't use too many basic types in a class.

```
Class Employee {
    private String street;
    private String city;
    private String state;
    private int zip;
}
```

→

```
Class Employee {
    private Address address;
}

class Address {
    private String street;
    private String city;
    private String state;
    private int zip;
}
```


90

Java Inheritance

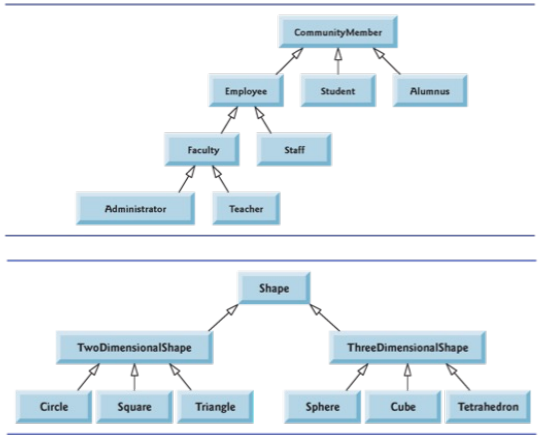
- Inheritance
  - Create a new class by acquiring an existing class's members and possibly enhancing it with new or modified capabilities.
  - Helps in reuse code.
  - Increases the likelihood that a system will be implemented and maintained effectively.

91

Java Inheritance

- Inheritance
  - Designate that the new class should inherit the members of an existing class.
    - Existing class is the **superclass**
    - New class is the **subclass**
  - A subclass is more specific than its superclass and represents a more specialized group of objects.
    - This is why inheritance is sometimes referred to as **specialization.**

92



93

Java Inheritance

- Inheritance
  - The direct superclass is the superclass from which the subclass explicitly inherits.
  - An indirect superclass is any class above the direct superclass in the class hierarchy.
  - The Java class hierarchy begins with class **Object** (in package java.lang)
    - Every class in Java directly or indirectly extends (or "inherits from") Object.
  - Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

94

Java Inheritance

- Class Inheritance
  - A class marked as “final” cannot be sub-classed or inherited from.
  - A class cannot inherit constructors
  - A superclass’s class initializers always execute before a subclass class initializers.
  - A subclass’s constructor always calls the superclass constructor to initialize an object’s superclass layer, and then initializes the subclass layer.

95

Java Inheritance

- Class Inheritance
  - Creating an inheritance relationship
    - “extends” keyword is used to specify inheritance relationship.

```
public class Point3D extends Point {
    int z = 10;

    public Point3D(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}
```


96

Java Inheritance

- Member access in Inheritance
  - superclass's public members are accessible by its subclasses.
  - superclass's protected members can be accessed by its subclasses
  - All public and protected superclass members retain their original access modifier when they become members of the subclass.

97

Java Inheritance

- Constructors
  - Calling base class constructor

```
public class Logger {

    byte[] buffer;

    public Logger() {
        super();
        buffer = new byte[1024];
    }

    public Logger(byte[] buf)
    {
        this();
        buffer = buf;
    }
```

Only available in Constructor

98 Timmins Training Consulting

98

Java Inheritance

- Method Overriding
  - All inherited methods can be overridden.
    - Except those are marked “final”.
  - Explicitly calling a base class method

```
public class Point3D extends Point {
    int z = 10;

    public Point3D(int a, int b, int c) {
        super(a, b);
        z = c;
    }

    @Override
    public String toString() {
        String str = super.toString();
        return str + " z = " + Integer.toString(z);
    }
```


99

Java Inheritance

- Object class
  - Super class of all classes in Java

`Object clone()`	Create and return a copy of the current object.
`boolean equals(Object obj)`	Determine if the current object is equal to the object identified by `obj`.
`void finalize()`	Finalize the current object.
`Class<?> getClass()`	Return the current object's `Class` object.
`int hashCode()`	Return the current object's hash code.
`void notify()`	Wake up one of the threads that are waiting on the current object's monitor.
`void notifyAll()`	Wake up all threads that are waiting on the current object's monitor.
`String toString()`	Return a string representation of the current object.

100

Java Inheritance

- Polymorphism
  - "Program in the general"
  - Treat objects in same class hierarchy as if all superclass objects
  - Abstract class
    - Common functionality
  - Makes programs extensible
    - New classes added easily, can still be processed

101

Java Inheritance

- Abstract classes
  - Are superclasses (called abstract superclasses)
  - Cannot be instantiated
  - Incomplete
    - subclasses fill in "missing pieces"
- Concrete classes
  - Can be instantiated
  - Implement every method they declare
  - Provide specifics

102

Java Inheritance

- Abstract Classes and Methods
  - Abstract classes often appear at the top of a hierarchy of classes
  - To make a class abstract
    - Declare with keyword **`abstract`**
    - Contain one or more *abstract methods*

```
public abstract void draw();
```

  - Abstract methods
    - No implementation, must be overridden

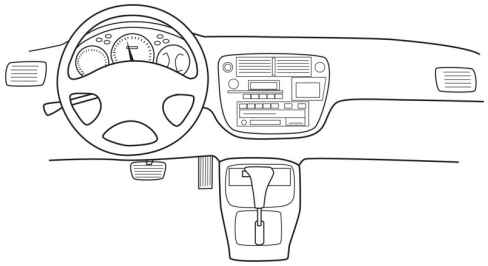
103

Java Inheritance

- `final` methods
  - Cannot be overridden in any subclass
  - `private` methods are implicitly `final`
  - `static` methods are implicitly `final`
- `final` classes
  - Cannot be superclasses
  - Methods in `final` classes are implicitly `final`
  - e.g., class `String`

104

Interfaces

- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
  - 

105

Interfaces

- Java Interface
  - Collection of abstract methods and constants that represent a functionality
  - An interface declaration begins with the keyword interface and contains only constants and abstract methods.
  - No implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly public abstract methods.
  - All fields are implicitly public, static and final.

106

Java Interfaces

- Defining an interface

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2 (float value, char ch);
    public boolean doTheOther (int num);
}
```


107

Java Interfaces

- Classes can implement an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface

```
public class CanDo implements Doable
{
    public void doThis ()
    {
    }

    public int doThat ()
    {
        return 1;
    }
……————
}
```


108

Java Interfaces

- A class can implement multiple interfaces
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
{
   // all methods of both interfaces
}
```


109

Java Interfaces

- Interface Inheritance
  - Interfaces can also be associated in hierarchical relationship

```
public interface BaseInterface {
    public void foo();
}

public interface SubInterface extends BaseInterface{
    public void bar();
}
```


110

Java Interfaces

- Interface Inheritance
  - Class implementing the Interface must implement all methods that exist in the hierarchy

```
public class MyClass implements SubInterface {
    @Override
    public void bar() {

    }

    @Override
    public void foo() {

    }
}
```


111

Java Interfaces

- Interface names can be used like class names in the parameters passed to a method

```
public boolean isLess(Comparable a, Comparable b) {
    return a.compareTo(b) < 0;
}
```

- Any class that "implements `Comparable`" can be used for the arguments to this method

112

Java Interfaces

- The Java standard library includes lots more built-in interfaces
- Examples:
  - `Clonable` – implements a `clone()` method
  - `Comparable` – implements a `compareTo()` method

113

Java Interfaces

- Example: Comparable Interface
  - Part of the java.lang package
  - Any class can implement Comparable to provide a mechanism for comparing objects of that type

```
public interface Comparable {
        public int compareTo(Object o);
}
```


114

Java Interfaces

- Iterator Interface
  - Provides a means of processing a collection of objects one at a time
  - Methods of Iterator interface
    - `hasNext` method returns a boolean result – true if there are items left to process
    - `next` method returns the next object in the iteration
    - `remove` method removes the object most recently returned by the `next` method

115

Java Interfaces

- Collection interface
  - General interface for any type that can store multiple values
  - Any object `c` that implements `Collections` has these methods
    - `c.add(e)`
    - `c.remove(e)`
    - `c.size()`

116

Java Interfaces

- Collection derived interfaces
  - `Set`: unordered, can't add the same object twice
  - `List`: ordered, adds new methods
    - `get(i)` : get the ith element
    - `set(i,e)` : set the ith element to `e`

117

Java Interfaces

- Implementation of Interfaces
  - List: `ArrayList, Stack, LinkedList`
  - Sets: `HashSet, TreeSet`
- Each implementation has some differences… suitable for particular problems
  - e.g. additional methods, different type restrictions, etc.

118

Java Interfaces

- Usage of Interfaces
  - Interfaces are a key aspect of object-oriented design in Java
  - Allows unrelated classes implement a set of common methods.
    - Example: Collections
  - Various design patterns follow an interface based approach.
  - Reduced concrete dependency between objects

119

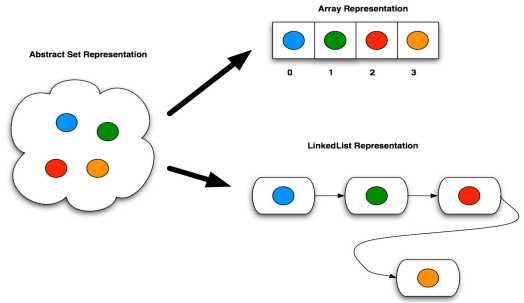
Java Interfaces

- Multiple Representation
  - When designing software, we too have a choice of representations
  - Suppose we want to model a set
    - Arrays
    - LinkedLists
    - Others?
  - How does this look conceptually?

120

Java Interfaces

- Multiple Representation



121

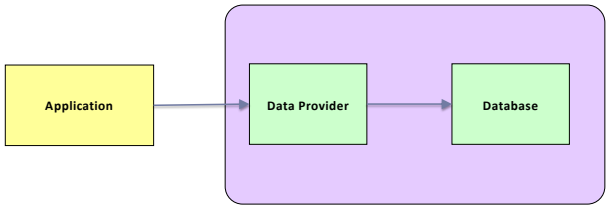
Java Interfaces

- Multiple Representation
  - Use an interface to capture the essential elements that must be common across all representations
  - Have concrete implementations (classes) implement the interface

122

Java Interfaces

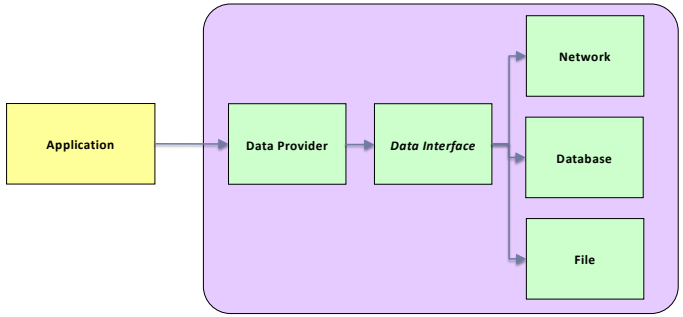
- Example: Data Processing



123

Java Interfaces

- Example: Data Processing



124

Java Interface

- Object Communication

```
public interface ButtonListener {
    public void userTapped(Button button);
}


public class Button {
    String title;
    ButtonListener listener;

    public void tapped() {
        listener.userTapped(this);
    }
}
```


125

Java Interface

- Object Communication

```
public class MyButtonListener implements ButtonListener{
    @Override
    public void userTapped(Button button) {
        System.out.println("Hello!!");
    }


public static void main(String[] args) {
    Button button = new Button();
    button.title = "Greeting";

    MyButtonListener listener = new MyButtonListener();
    button.listener = listener;

    button.tapped();
}
```


126

Anonymous Classes

Amit Gulati
amit.gulati@gmail.com

127

Anonymous Class

- Class without a name.
- Simultaneously declared and instantiated any place where it is legal to specify an expression.

```
public class ACDemo
{
   public static void main(final String[] args)
   {
      new Object()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to do";
         void dosomething()
         {
            System.out.println(msg);
         }
      }
      .dosomething();
   }
}
```


128

Anonymous Classes

- May inherit from classes, and also implement interfaces.
  - Anonymous class inheriting from another class

```
abstract class Speaker
{
   abstract void speak();
}

public class ACDemo1
{
   public static void main(final String[] args)
   {
      new Speaker()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";
         void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```


129

Anonymous Classes

- May inherit from classes, and also implement interfaces.
  - Anonymous class implementing Interface

```
interface Speaker
{
   void speak();
}

public class ACDemo1
{
   public static void main(final String[] args)
   {
      new Speaker()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";

         public void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```


130

Anonymous Classes

- Do not have constructors, but call the base class constructors after the new operator.
- Anonymous class instances should be able to access the surrounding scope's local variables and parameters.
  - Only if they are market "final".

131

Lambda Expression

- Lambda expression denotes a piece of functionality
  - A block of code that you can pass around so it can be executed later, once or multiple times.

```
(int x) -> { return x + 1; };
```

- Alternative to anonymous class for interfaces with just one method
  - Single Abstract Method (SAM) interfaces

132

Lambda Expression

- Syntax
  - parameter list followed by the arrow token and a body, as in:
    - (parameterList) -> {statements}

```
(int x, int y) -> {return x + y;}
```

  - parameter types may be omitted

```
(x, y) -> {return x + y;}
```

  - return types are determined by the lambda's context.

133

Lambda Expression

- Syntax
  - Lambda expression with parameters and single line of code

```
(String first, String second)
        -> first.length() - second.length();
```

  - Lambda with multiple lines of code

```
(String first, String second) ->
{
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
};
```


134

Lambda Expression

- Syntax
  - Lambda expression with no parameters

```
() -> {
    for (int i = 100; i >= 0; i--)
       System.out.println(i);
}
```


135

Nested Types

Amit Gulati
amit.gulati@gmail.com

136

Introduction

- Top-Level classes
  - Classes that are declared outside of any class are known as *top-level classes*.
- Inner classes or Nested Types
  - In OO code, there may be a need for creating classes that are declared inside other classes.
  - Java programming language allows you to define a class within another class, known as *Inner classes* or *Nested Classes* or *Nested Types*.

```
class OuterClass {
    ...
    …
    class InnerClass {
        ...
    }
}
```


137

Introduction

- Inner Class
  - A nested class is a member of its enclosing class.
  - A nested class can be declared private, public, protected, or *package private.*
  - Inner classes are divided into two categories
    - Non-static
    - Static

138

Introduction

- Inner Class
  - Non-Static Inner Class
    - Non-static Inner classes have access to other members of the enclosing class, even if they are declared private.

```
class OuterClass {
   ...
   …
   class InnerClass {
      ...
   }
}
```

Non-static Inner class

139 Timmins Training Consulting

139

Inner Classes

- Inner Class
  - Static Inner Classes
    - Classes that are marked with the keyword "static", are static inner classes.

```
class OuterClass {
   ...
   …
   class InnerClass {
      ...
   }
   static class StaticInnerClass {
   {
      ....
    }
}
```

Static Inner class

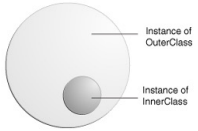
- Static nested classes do not have access to other members of the enclosing class.

140 Timmins Training Consulting

140

Inner Classes

- Inner Class
  - An instance of InnerClass can exist only within an instance of OuterClass.



  - To instantiate an inner class, you must first instantiate the outer class.

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```


141

Inner Classes

- Variable and Method Access
  - Non-static inner class
    - Access to all members (instance and class) of outer class, even if they are private.
  - Static inner class
    - Access to all class (static) members, even if they are declared private.
  - **this** reference
    - Inside Inner class, **this** reference points to the Inner instance.
    - To refer to **this** reference of outer instance, prefix **this** with the outer class name.

142

Error Handling/Exceptions

Amit Gulati
amit.gulati@gmail.com

143

Introduction

- Errors
  - Unexpected result obtained from a program during execution.
  - Programmer Error
    - Incorrect logic
    - Division by zero
    - Array access
    - Illegal Cast
    - Etc.
  - Runtime System Condition
    - Out of memory
    - File not available
    - Network not available
    - Etc.

144

Introduction

- Error
  - Unhandled errors may manifest themselves as incorrect results or behavior, or as abnormal program termination.
  - Programmer Responsibility
- Error Handling Approaches
  - Traditional Error Handling
    - Using return values
    - Using jump instruction (jump to error handling code)
  - Exception handler

Introduction

- Traditional Error Handling
  - Example : Error checking code mixed with code logic

```
int readFile() {
    int errorCode = 0;
    //open the file;
    if (theFileIsOpen) {
        //determine the length of the file;
        if (gotTheFileLength) {
            //allocate that much memory;
            if (gotEnoughMemory) {
                //read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
            return errorCode;
        }
    }
}
```

Exceptions

- Exceptions
  - Modern Approach to error handling.
  - Default way of handling errors in a Java program.
  - Advantages of Exception
    - Exception Handling code block separate from the program logic
    - Exceptions can propagate to higher contexts.
    - OO approach to Error handling
      - Exception is an Object and not just an error code.
  - Disadvantages of Exceptions
    - Exception Handling is slow.

Exceptions

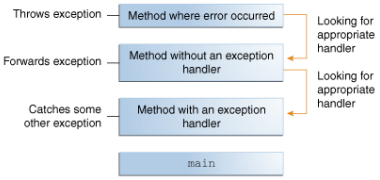
- Exception Mechanism
  - An exception object is a representation of an error condition.

```
class ExceptionHandling {
    public static void main(String[] args) {
        printAverage(100, 0);
    }

    public static void printAverage(int totalSum, int totalCount) {
        int average = computeAverage(totalSum, totalCount);
    }

    public static int computeAverage(int sum, int count) {
        System.out.println("Computing average.");
        return sum/count;
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionHandling.computeAverage(ExceptionHandling.java:13)
        at ExceptionHandling.printAverage(ExceptionHandling.java:8)
        at ExceptionHandling.main(ExceptionHandling.java:4)
```

Exceptions

- Exception Mechanism
  - When an error occurs within a method
    - After a method throws an exception, the runtime system attempts to find something to handle it.
    - The runtime system searches the call stack for a method that contains a block of code that can handle the exception, known as *exception handler*.



149

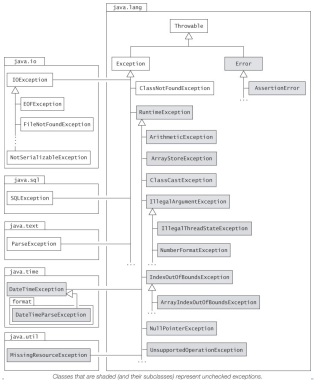
Exceptions

- Exceptions can Broadly be classified into 2 categories
  - Checked Exceptions
  - Un-Checked Exceptions
- Checked Exceptions
  - Exceptional conditions that a well-written application should anticipate and recover from.
  - All exceptions are Checked Exceptions
    - Except Exception classes that inherit from **Error** and **RuntimeException**.
  - if a method can throw a checked exception, the method must either catch the exception and take the appropriate action.

151

Exceptions

- Exception Class Heirarchy



Exceptions

- Creating a custom Exception class
  - New customized exceptions are usually defined by either extending the **Exception** class or one of its checked subclasses
  - Customized exceptions, as any other Java classes, can declare fields, constructors, and methods, thereby providing more information as to their cause

```
public class EvacuateException extends Exception {
    private Date date;
    private Zone zone;
    private TransportMode transport;
    public EvacuateException(Date d, Zone z, TransportMode t) {
        super("Evacuation of zone " + z);
    }
}
```

Exceptions

- Exception Handling
  - If your code is calling a method that might throw exceptions
    - Enclose the method call in try-catch statement and handle the Exception thrown in the catch block.

```
try {
    // program logic
    //
    method that throws exception


}
catch (Exception e) {
    //error handling code
}
```


154

Exceptions

- Exception Handling
  - If your code is calling a method that might throw exceptions
    - Enclose the method call in try-catch statement and handle the Exception thrown in the catch block.

```
public class ExceptionHandling {
    public static void main(String[] args) {
        FileInputStream fis = openFile("hello.txt");
    }

    public static FileInputStream openFile(String str) {
        try {
            FileInputStream fis = new FileInputStream(str);
            return fis;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.out.println("File named " + str + " not found");
            return null;
        }
    }
}
```


155

Exceptions

- Exception Handling
  - If your code is calling a method that might throw exceptions
    - Specify **throws** clause for the method, that list the Exceptions not being handled.

```
public class ExceptionHandling {
    public static void main(String[] args) {
        try {
            FileInputStream fis = openFile("hello.txt");
        }
        catch(FileNotFoundException e) {
            System.out.println("File not found");
        }
    }

    public static FileInputStream openFile(String str)
                                    throws FileNotFoundException {
        FileInputStream fis = new FileInputStream(str);
        return fis;
    }
}
```


Exceptions

- Exception Handling
  - Exceptions cause normal flow of the program to be disrupted.

Normal Flow

```
try {
    // program logic
    //
    //

}
catch (Exception e) {
    //error handling code
}
```

Exception

```
try {
    //program logic

    // error
    //
    //
}
catch (Exception e) {
    //error handling code
}
```

Exceptions

- Exception Handler
  - Three components of Exception Handler
    - **try** block
    - **catch** block
    - **finally** block

158

Exceptions

- Exception Handling
  - **try** block
    - Enclose the code that might throw an exception within a try block.
    - **try** block looks like the following

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```


159

Exceptions

- Exception Handling
  - **catch** block
    - Defines the block of code to be executed in case exception occurs
    - A try block must have at-least one catch block.
    - No code can be between the end of the try block and the beginning of the first catch block.

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```


160

Exceptions

- Exception Handling
  - **catch** block
    - There can be more than one catch blocks for a try block, each for a different type of Exception.

```
try {
    Thread.sleep(10000);
    FileInputStream fis = new FileInputStream("hello.text");
    fis.read();
} catch (InterruptedException e) {
    throw new RuntimeException(e);
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```


161

Exceptions

- Exception Handling
  - **catch** block
    - The runtime system invokes the exception handler when the handler is the first one in the call stack whose *ExceptionType* matches the type of the exception thrown.
    - The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

162

Exceptions

- Exception Handling
  - **catch** block
    - Example: FileInputStream throws a FileNotFoundException

```
public class Test {
    public static void main(String[] args)
    {
        try {
            FileInputStream fstr = new FileInputStream("hello.txt");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

    - The catch block specifying “Exception” will execute, as Execute is the base class for all exceptions.

163

Exceptions

- Exception Handling
  - **catch** block
    - Example: FileInputStream throws a FileNotFoundException

```
public class Test {
    public static void main(String[] args)
    {
        try {
                FileInputStream fstr = new FileInputStream("hello.txt");
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
            catch(Exception e) {
                e.printStackTrace();
            }
    }
}
```

    - The catch block specifying "FileNotFoundException" will execute in case of FileNotFoundException.

Exceptions

- Exception Handling
  - **catch** block
    - Alternately you can have many try catch blocks for different types of exceptions.

```
try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
try {
    FileInputStream fis = openFile("hello.txt");
}
catch(FileNotFoundException e) {
    System.out.println("File not found");
}
```

Exceptions

- Exception Handling
  - **finally** block
    - finally block *always* executes when the try block exits.

Normal Flow

```
try {
    // program logic
    //
    //
}
catch (Exception e) {
    //error handling code
}
finally {
    // do something
}
```

Exception

```
try {
    //program logic

    // exception
    //
}
catch (Exception e) {
    //error handling code
}
finally {
    //do something
}
```


Exceptions

- Exception Handling
  - **finally** block
    - finally block is a key tool for preventing resource leaks.
    - When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is *always* recovered.

- If the JVM exits while the try or catch code is being executed, then the finally block may not execute.
- If the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

Exceptions

- Passing Exceptions to Higher Context
  - Sometimes its better to let a method further up the call stack handle the exception.
  - .

```
public class Test {
    public static void main(String[] args)
    {
        try {
            FileInputStream str = openFile("hello.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static FileInputStream openFile(String str)
                                    throws FileNotFoundException
    {
        FileInputStream fstr = new FileInputStream(str);
        return fstr;
    }
}
```


168

Exceptions

- Rethrow Exceptions
  - Exceptions can be re-thrown from the try-catch block, in case we need to notify higher contexts.

```
public class Test {
    public static void main(String[] args)
    {
        try {
            FileInputStream str = openFile("hello.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static FileInputStream openFile(String str) throws FileNotFoundException
    {
        FileInputStream fstr = null;
        try {
            fstr = new FileInputStream(str);
        }
        catch (FileNotFoundException e) {
            System.out.println("Unable to Open File");
            throw e;
        }
        return fstr;
    }
}
```


169

Exceptions

- Throwing Exception
  - Before Exceptions can be caught in your program, they must be thrown.
  - Java Core Library methods throw exceptions and list the exceptions thrown in their method signatures.
  - **throw** statement is used for throwing Exceptions.
    - **throw** statement requires a single argument, a **throwable** object.
    - Throwable objects are instances of any subclass of the **Throwable** class.

```
throw new FileNotFoundException("Unable to Open File");
```