

# Java

*Package, Interface & Exception*

# ***Package***

# Package

- Java package provides a mechanism for partitioning the class name space into more manageable chunks
  - Both **naming** and **visibility** control mechanism
- Define classes inside a package that are not accessible by code outside that package
- Define class members that are exposed only to other members of the same package
- This allows classes to have intimate knowledge of each other
  - Not expose that knowledge to the rest of the world

# Declaring Package

- ***package pkg***
  - Here, pkg is the name of the package
- ***package mypackage***
  - creates a package called mypackage
- The package statement defines a name space in which classes are stored
- If you omit the package statement, the class names are put into the **default package**, which has no name

# Declaring Package

- Java uses file system directories to store packages
  - the .class files for any classes that are part of mypackage must be stored in a directory called mypackage
- More than one file can include the same package statement
- The package statement simply specifies to which package the classes defined in a file belong
- To create hierarchy of packages, separate each package name from the one above it by use of a (.)

# Package Example

```
1  package mypackage;
2
3  class Balance {
4      String name;
5      double bal;
6      Balance(String n, double b) {
7          name = n;
8          bal = b;
9      }
10     void show() {
11         System.out.println(name + ": $" + bal);
12     }
13 }
14 public class AccountBalance {
15     public static void main(String[] args) {
16         Balance [] current = new Balance[3];
17         current[0] = new Balance(n: "K. J. Fielding", b: 123.23);
18         current[1] = new Balance(n: "Will Tell", b: 157.02);
19         current[2] = new Balance(n: "Tom Jackson", b: -12.33);
20         for (Balance b : current) {
21             b.show();
22         }
23     }
24 }
```

**javac -d . AccountBalance.java**

**java mypackage.AccountBalance**

# Package Syntax

- The general form of a multilevel package statement
  - ***package pkg1[.pkg2[.pkg3]]***
  - ***package java.util.concurrent***
- import statements occur immediately following the package statement and before any class definitions
- The general form of the import statement
  - ***import pkg1 [.pkg2].(classname | \*)***
  - ***import java.util.Scanner***
  - import statement is optional, class can be used with name that includes full package hierarchy

# Access Protection

- Packages act as containers for classes and other subordinate packages
- Classes act as containers for data and code
- The class is Java's smallest unit of abstraction
- Four categories of visibility for class members
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different package
  - Classes that are neither in the same package nor subclasses



# Access Protection

- The three access modifiers provide a variety of ways to produce the many levels of access required
  - private, public, and protected
- The following applies only to members of classes

	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

For detail example, please refer to codes in package **p1** and **p2**

# Access Protection

- Anything declared ***public*** can be accessed from anywhere
- Anything declared ***private*** cannot be seen outside of its class
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package (***default access***)
- If you want to allow an element to be seen outside your current package, but only to classes that subclass the class directly, declare that ***protected***

# Access Protection

- A non-nested class has only two possible access levels
  - **default** and **public** (others are **abstract** and **final**)
- When a class is declared as public, it is accessible by any other code
- If a class has default access, then it can only be accessed by other code within its same package
- When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class

# ***Interface***

# Interface

- We can call it a pure abstract class having no concrete methods
  - All methods declared in an interface are implicitly **public** and **abstract**
  - All variables declared in an interface are implicitly **public**, **static** and **final**
- *An interface can't have instance variables, so can't maintain state information unlike class*
- A class can only extend from a **single class**, but a class can implement **multiple interfaces**

# Implementing Interface

- When you implement an interface method, it must be declared as public
- By implementing an interface, a class signs a contract with the compiler that it will definitely provide implementation of all the methods
  - If it fails to do so, the class will be considered as abstract
  - Then it must be declared as abstract and no object of that class can be created
- An abstract class specifies **what an object is** and an interface specifies **what the object can do**

# Simple Interface

```
1 interface Callback {
2     void call(int param);
3 }
4
5 class Client implements Callback {
6     public void call(int p) {
7         System.out.println("call method called with " + p);
8     }
9     public void f() {
10        System.out.println("simple method, not related with Callback");
11    }
12 }
13 public class InterfaceTest {
14     public static void main(String[] args) {
15         // Error, Callback is abstract, can't be instantiated
16         // Callback c = new Callback();
17         // Can't instantiate an interface directly
18         Client client = new Client();
19         client.call(p: 42);
20         client.f();
21         // Accessing implementations through Interface reference
22         Callback cb = new Client();
23         cb.call(param: 84);
24         // cb.f(); Error, no such method in Callback
25     }
26 }
```

# Simple Interface

```
1 interface Callback {  
2     void call(int param);  
3 }  
4  
5 public class InterfaceTest {  
6     public static void main(String[] args) {  
7         // Anonymous class that implements Callback, introduced in Java 8  
8         Callback callback = new Callback() {  
9             @Override  
10            public void call(int param) {  
11                System.out.println("call method called with " + param);  
12            }  
13        };  
14        callback.call(param: 10);  
15    }  
16 }
```



# Applying Interfaces

```
1  interface MyInterface {
2      void print(String msg);
3  }
4
5  class MyClass1 implements MyInterface {
6      public void print(String msg) {
7          System.out.println(msg + ":" + msg.length());
8      }
9  }
10
11 class MyClass2 implements MyInterface {
12     public void print(String msg) {
13         System.out.println(msg.length() + ":" + msg);
14     }
15 }
16 public class InterfaceApplyTest {
17     public static void main(String[] args) {
18         MyClass1 mc1 = new MyClass1();
19         MyClass2 mc2 = new MyClass2();
20         MyInterface mi; // create an interface reference variable
21         mi = mc1;
22         mi.print("Hello World");
23         mi = mc2;
24         mi.print("Hello World");
25     }
26 }
```

# Nested or Member Interfaces

```
1  class A {  
2      // non-nested interfaces can be default or public  
3      // nested interfaces can be private/protected/public/default  
4      interface NestedIF {  
5          boolean isNonNegative(int x);  
6      }  
7  }  
8  
9  class B implements A.NestedIF {  
10     public boolean isNonNegative(int x) { return x >= 0; }  
13 }  
14 public class InterfaceNestedTest {  
15     public static void main(String[] args) {  
16         A.NestedIF nif = new B();  
17         System.out.println(nif.isNonNegative(x: 100));  
18         System.out.println(nif.isNonNegative(x: -10));  
19     }  
20 }
```

# Variables in Interfaces

```
1  import java.util.Random;
2
3  interface SharedConstants {
4      int NO = 1;
5      int YES = 2;
6  }
7
8  class Question implements SharedConstants {
9      Random rand = new Random();
10     int ask() {
11         int prob = (int) (100 * rand.nextDouble());
12         if (prob < 50) return NO;
13         else return YES;
14     }
15 }
16 public class InterfaceVariableTest {
17     public static void main(String[] args) {
18         Question q = new Question();
19         for (int i = 0; i < 10; i++) {
20             System.out.println(q.ask());
21         }
22     }
23 }
24
```

# Extending Interfaces

```
1 interface I1 {  
2     void f1();  
3 }  
4 interface I2 {  
5     void f2();  
6 }  
7 interface I3 extends I1, I2 {  
8     void f3();  
9 }  
10 class MyClass implements I3 {  
11     public void f1() { System.out.println("Implement f1"); }  
14     public void f2() { System.out.println("Implement f2"); }  
17     public void f3() { System.out.println("Implement f3"); }  
20 }  
21  
22 public class InterfaceExtendsTest {  
23     public static void main(String[] args) {  
24         MyClass m = new MyClass();  
25         m.f1();  
26         m.f2();  
27         m.f3();  
28     }  
29 }
```

# Default Interface Methods

- Prior to Java 8, an interface could not define any implementation whatsoever
- The release of Java 8 has changed this by adding a new capability to interface called the *default method*
  - A default method lets you define a default implementation for an interface method
  - Its primary motivation was to provide a means by which interfaces could be expanded without breaking existing code

# Default Interface Methods

```
1 interface MyIF {  
2     // This is a "normal" interface method declaration.  
3     int getNumber();  
4     // This is a default method. Notice that it provides  
5     // a default implementation.  
6     default String getString() { return "Default String"; }  
9 }  
10  
11 class MyIFImp implements MyIF {  
12     // Only getNumber() defined by MyIF needs to be implemented.  
13     // getString() can be allowed to default.  
14     public int getNumber() { return 100; }  
17 }  
18  
19 public class InterfaceDefaultMethodTest {  
20     public static void main(String[] args) {  
21         MyIFImp m = new MyIFImp();  
22         System.out.println(m.getNumber());  
23         System.out.println(m.getString());  
24     }  
25 }  
26
```

# Multiple Inheritance Issues

```
3 interface Alpha {
4     default void reset() {
5         System.out.println("Alpha's reset");
6     }
7 }
8
9 interface Beta {
10    default void reset() {
11        System.out.println("Beta's reset");
12    }
13 }
14
15 class TestClass implements Alpha, Beta {
16     public void reset() {
17         System.out.println("TestClass's reset");
18     }
19 }
```

```
3 interface Alpha {
4     default void reset() {
5         System.out.println("Alpha's reset");
6     }
7 }
8
9 interface Beta extends Alpha {
10    default void reset() {
11        System.out.println("Beta's reset");
12        // Alpha.super.reset();
13    }
14 }
15
16 class TestClass implements Beta {
17
18 }
```

# Static Methods in Interface

```
1  interface MyIFStatic {  
2      int getNumber();  
3  
4      default String getString() {  
5          return "Default String";  
6      }  
7  
8      // This is a static interface method (introduced in Java 8)  
9      // not inherited by either an implementing class or a subinterface.  
10     static int getDefaultNumber() {  
11         return 0;  
12     }  
13 }  
14  
15 ▶ public class InterfaceStaticMethodTest {  
16 ▶     public static void main(String[] args) {  
17         System.out.println(MyIFStatic.getDefaultNumber());  
18     }  
19 }
```



# Private Methods in Interface

```
1 interface MyIFPrivate {  
2     default String f1() {  
3         login();  
4         return "Hello";  
5     }  
6     default String f2() {  
7         login();  
8         return "World";  
9     }  
10    // This is a private interface method (introduced in Java 9)  
11    // can be called only by a default method or another private method of the same interface  
12    private void login() {  
13        System.out.println("login");  
14    }  
15 }  
16 class MyIFPrivateImp implements MyIFPrivate {  
17 }  
18 public class InterfacePrivateMethodTest {  
19     public static void main(String[] args) {  
20         MyIFPrivate ifp = new MyIFPrivateImp();  
21         System.out.println(ifp.f1());  
22         System.out.println(ifp.f2());  
23     }  
24 }
```

# ***Exception***

# Exception Handling

- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
  - That method may choose to handle the exception or pass it on (caught and processed at some point)
- Generated by the Java runtime or by your code
  - Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment
  - Manually generated exceptions are typically used to report some error condition to the caller of a method

# Exception Handling

- Java exception handling is managed via five keywords
  - Program statements that you want to monitor for exceptions are contained within a **try** block
  - If an exception occurs within the try block, it is thrown
  - Your code can catch this exception (using **catch**)
  - To manually throw an exception, use the keyword **throw**
  - Any exception that is thrown out of a method must be specified as such by a **throws** clause
  - Any code that absolutely must be executed after a try block completes is put in a **finally** block

# Exception Classes Hierarchy

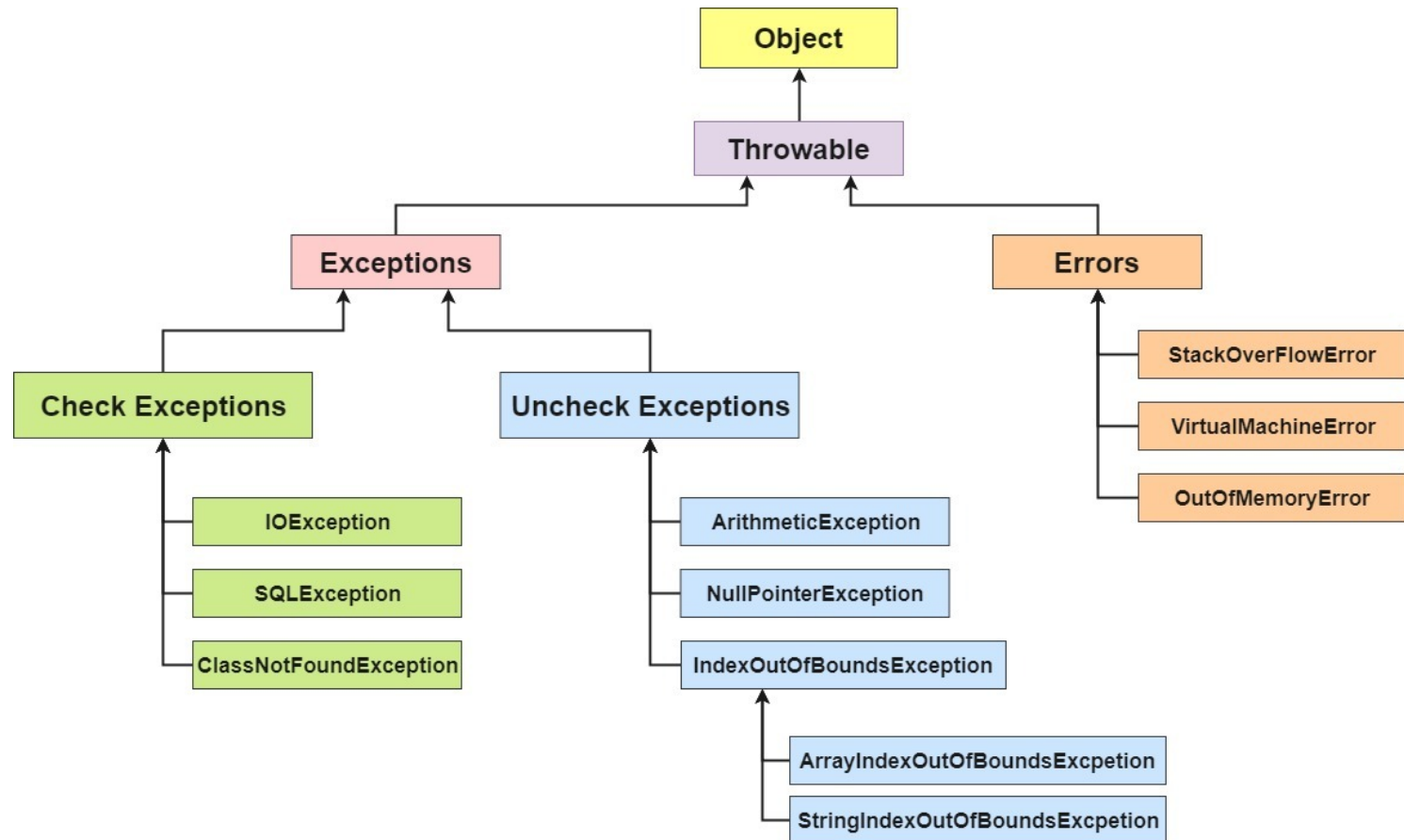


Image Source: <https://simplesnippets.tech/exception-handling-in-java-part-1/>

Complete List of Java Exceptions: <https://programming.guide/java/list-of-java-exceptions.html>

# Uncaught Exceptions

```
1 public class ExceptionUncaught {  
2     public static void main(String args[]) {  
3         int a = 10, b = 0;  
4         int c = a / b; // ArithmeticException: / by zero  
5         System.out.println(a);  
6         System.out.println(b);  
7         System.out.println(c);  
8         String s = null;  
9         System.out.println(s.length()); // NullPointerException  
10    }  
11 }
```

# Caught Exceptions

```
1 ▶ public class ExceptionCaught1 {
2 ▶     public static void main(String args[]) {
3         int a = 10, b = 0, c = 0;
4         try {
5             // try requires at least one catch or a finally clause
6             c = a / b;
7             System.out.println("This will never print");
8         } catch (Exception e) { // ArithmeticException
9             System.out.println("In Catch");
10            System.out.println(e);
11        } finally {
12            // finally block is optional
13            // finally block will always execute
14            System.out.println("In Finally");
15        }
16        System.out.println(a);
17        System.out.println(b);
18        System.out.println(c);
19    }
20 }
```

# Caught Exceptions

```
1  import java.util.Random;
2
3  ▶ public class ExceptionCaught3 {
4  ▶  ▶ public static void main(String args[]) {
5      int a = 10, b, c;
6      Random r = new Random();
7      for (int i = 1; i <= 32000; i++)
8      {
9          try {
10             b = r.nextInt();
11             c = r.nextInt();
12             a = 12345 / (b / c);
13         } catch (ArithmeticException e) {
14             System.out.println(e);
15             a = 0;
16         } finally {
17             System.out.println(i + ": " + a);
18         }
19     }
```

try can be nested, please refer to **ExceptionTryNested.java**



# finally

```
1  ▶ public class ExceptionCaught2 {  
2  ▶  public static void main(String args[]) {  
3      int a = 10, b = 0, c;  
4      try {  
5          c = a / b;  
6          System.out.println("This will never print");  
7      } catch (Exception e) { // ArithmeticException  
8          System.out.println("In Catch");  
9          System.out.println(e);  
10         return;  
11     } finally {  
12         // finally block will always execute  
13         System.out.println("In Finally");  
14     }  
15     System.out.println(a);  
16     System.out.println(b);  
17     System.out.println(c);  
18 }  
19 }
```

# Multiple catch clauses

```
1  ▶ public class ExceptionMultipleCatch {
2  ▶  public static void main(String args[]) {
3      int a = 10, b = 0, c = 0;
4      try {
5          c = a / b;    catch(ArithmeticException | NullPointerException e)
6      } catch (ArithmeticException e1) {
7          System.out.println(e1);
8      } catch (NullPointerException e2) {
9          System.out.println(e2);
10     } catch (Exception e) {
11         System.out.println(e);
12     } finally {    catch(ArithmeticException | Exception e) - Error
13         System.out.println("In Finally");
14     }
15     System.out.println(a);
16     System.out.println(b);
17     System.out.println(c);
18 }
19 }
```

# throw

```
1 ► public class ExceptionThrow {  
2     public static void f() {  
3         try {  
4             throw new NullPointerException("f");  
5         } catch(NullPointerException e) {  
6             System.out.println("Inside catch of f()");  
7             throw e; //rethrow the exception  
8         }  
9     }  
10 ► public static void main(String args[]) {  
11     try {  
12         f();  
13     } catch(NullPointerException e) {  
14         System.out.println("Inside catch of main()");  
15     }  
16 }  
17 }
```

# throws

```
1  ► public class ExceptionThrows {  
2      public static void f() throws IllegalAccessException {  
3          System.out.println("Inside f()");  
4          throw new IllegalAccessException("f");  
5      }  
6  
7  ► public static void main(String args[]) {  
8      try {  
9          f();  
10     } catch (IllegalAccessException e) {  
11         System.out.println("Inside catch of main()");  
12         e.printStackTrace();  
13     }  
14 }  
15 }
```

throws listing is not required for those of RuntimeException or any of their subclasses

# Custom Exceptions

```
1  class MyException extends Exception {  
2      private int detail;  
3      MyException(int a) { detail = a; }  
6      @Override  
7      public String toString() { return "My Exception : " + detail; }  
10 }  
11  
12 public class ExceptionCustom {  
13     static void compute(int a) throws MyException {  
14         if (a > 10) {  
15             throw new MyException(a);  
16         }  
17         System.out.println(a);  
18     }  
19     public static void main(String args[]) {  
20         try {  
21             compute(a: 10);  
22             compute(a: 20);  
23         } catch (MyException e) {  
24             System.out.println(e);  
25         }  
26     }  
27 }
```