Module 8

Exceptions and Assertions

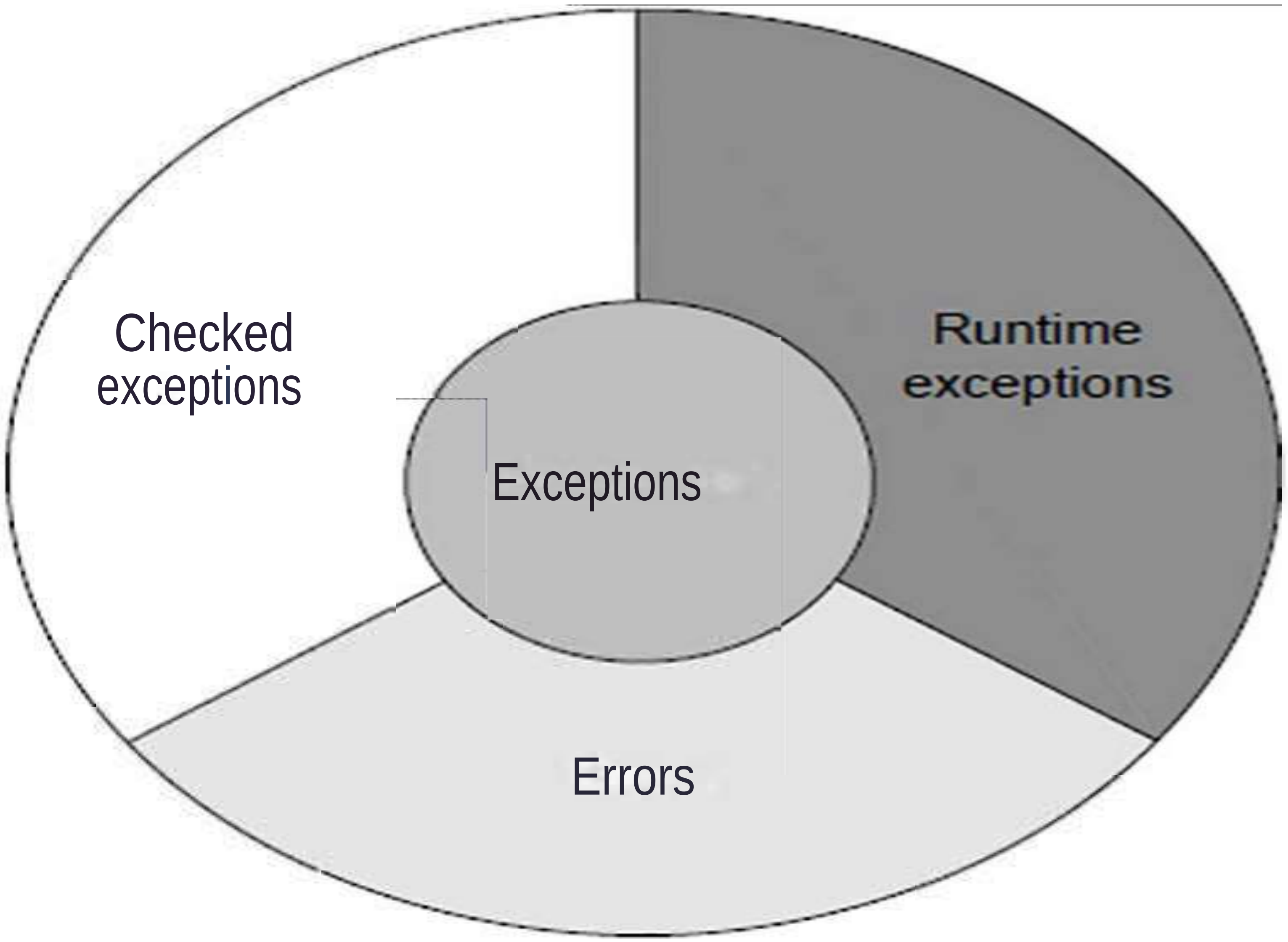
Objectives

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions*
- Log*
- Annotation*

Exceptions

Exceptions

- Separates the code that deals with errors from the code.
- An exception is an object that's created when an abnormal situation arises. This object has fields that store information about the nature of the problem.
- Conditions that can readily occur in a correct program are *checked exceptions*.
  These are represented by the Exception class.
- Severe problems treated as fatal or situations that reflect program bugs are *unchecked exceptions*.
  Fatal situations are represented by the Error class. Probable bugs are represented by the RuntimeException class.



Unchecked exceptions = Runtime exception + Errors

Exception Example

```
1   public class AddArguments {
2     public static void main(String args[]) {
3       int sum = 0;
4       for ( String arg : args )
5         { sum +=
6         Integer.parseInt(arg);
7       }
8       System.out.println("Sum = " + sum);
9   }
```

```
java AddArguments 1 2 3 4
Sum = 10
```

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

The `try-catch` Statement

```
public class AddArguments2 {
  public static void main(String args[]) {
    try {
      int sum = 0;
      for ( String arg : args )
        { sum +=
        Integer.parseInt(arg);
      }
    } catch (NumberFormatException nfe) {
      System.err.println("One of the command-line "
                         + "arguments is not an integer.");
    }
  }
}
```

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

The `try-catch` Statement

```java
public class AddArguments3 {
  public static void main(String args[])
    { int sum = 0;
    for ( String arg : args ) {
      try {
        sum += Integer.parseInt(arg);
      } catch (NumberFormatException nfe) {
        System.err.println("[" + arg + "] is not an integer"
                                   + " and will not be included in the sum.");
      }
    }
    System.out.println("Sum =  " + sum);
  }
}
```

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

The `try-catch` Statement

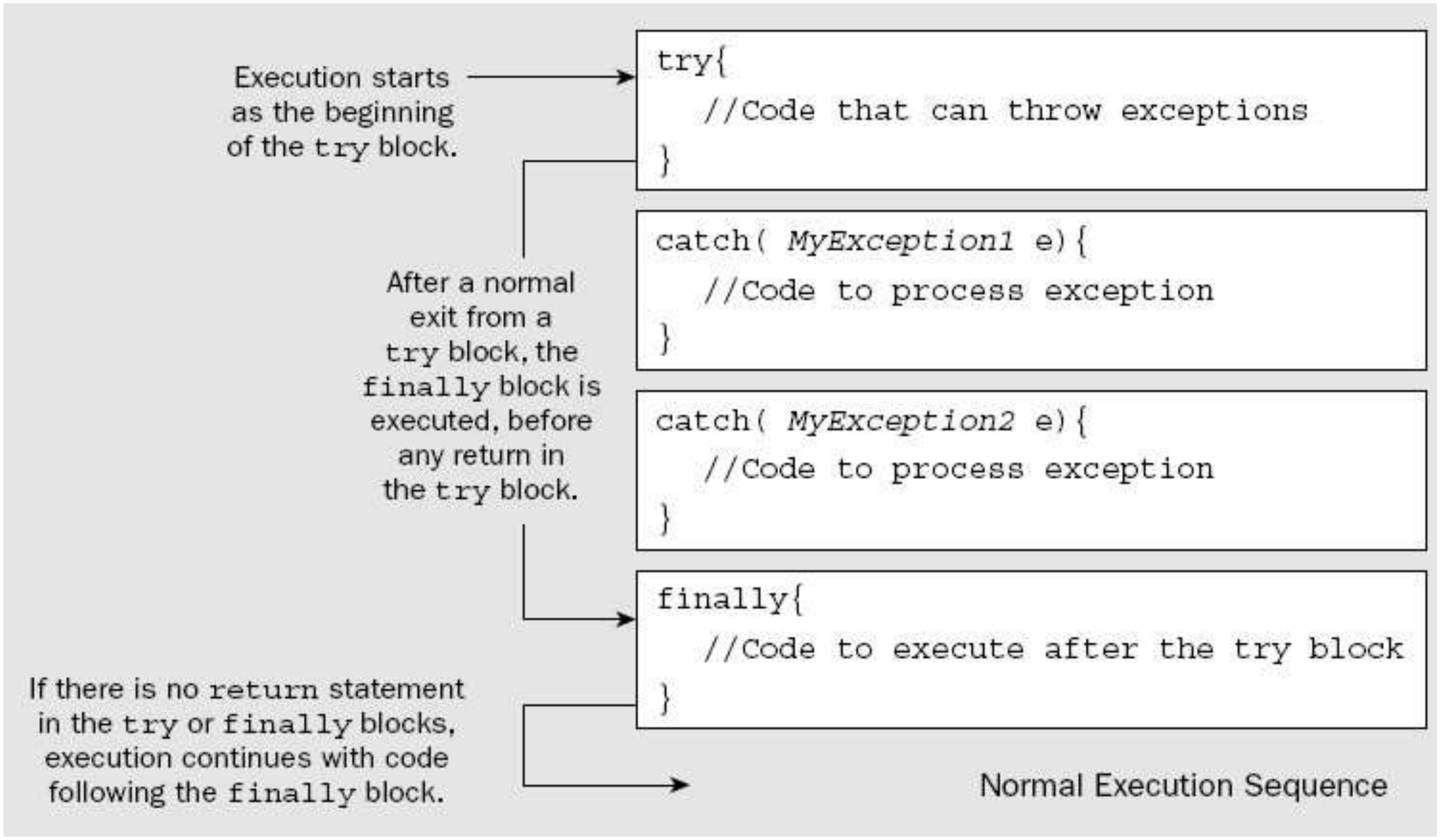
A `try-catch` statement can use multiple catch clauses:

```
try {
  // code that might throw one or more exceptions
} catch (MyException e1) {
  // code to execute if a MyException exception is thrown

} catch (MyOtherException e2) {
  // code to execute if a MyOtherException exception is thrown

} catch (Exception e3) {
  // code to execute if any other exception is thrown
}
```

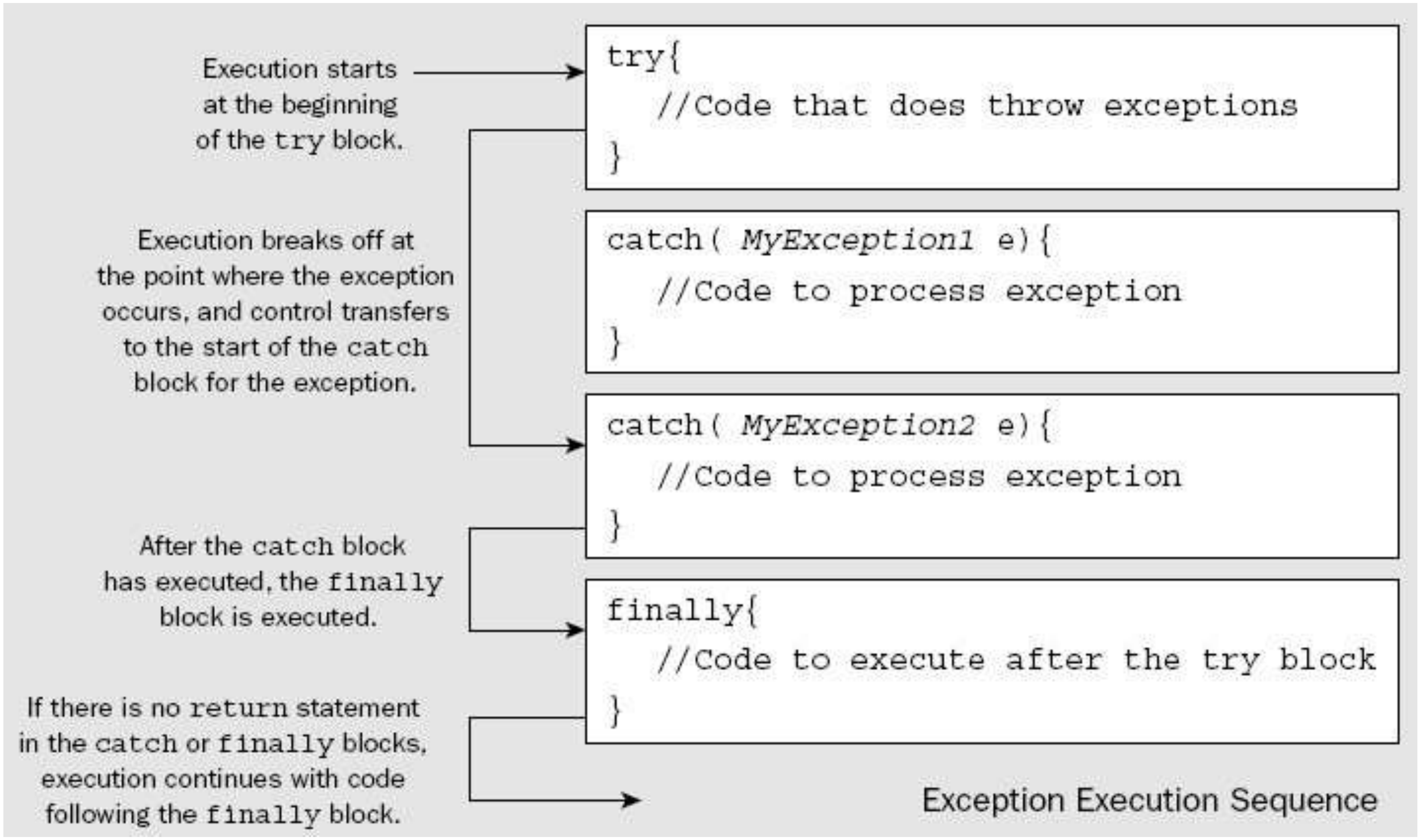
The `finally` Clause

The `finally` clause defines a block of code that *always* executes.

```
try {
  startFaucet();
  waterLawn();
} catch (BrokenPipeException e) {
  logProblem(e);
} finally {
  stopFaucet();
}
```


Execution starts as the beginning of the try block.
After a normal exit from a try block, the finally block is executed, before any return in the try block.
If there is no return statement in the try or finally blocks, execution continues with code following the finally block.
Normal Execution Sequence

```
try{
   //Code that can throw exceptions
}
catch( MyException1 e){
   //Code to process exception
}
catch( MyException2 e){
   //Code to process exception
}
finally{
   //Code to execute after the try block
}
```


Execution starts at the beginning of the try block.
Execution breaks off at the point where the exception occurs, and control transfers to the start of the catch block for the exception.
After the catch block has executed, the finally block is executed.
If there is no return statement in the catch or finally blocks, execution continues with code following the finally block.
Exception Execution Sequence

```
try{
   //Code that does throw exceptions
}
```

```
catch( MyException1 e){
   //Code to process exception
}
```

```
catch( MyException2 e){
   //Code to process exception
}
```

```
finally{
   //Code to execute after the try block
}
```

ATest: TestFinally.java

try-with-resources

- Resource-release code should be placed in a finally block
- *try-with-resources* statement (since JDK 7) simplifies release resources.But each resource must implements the *AutoCloseable* interface: *close()* method.

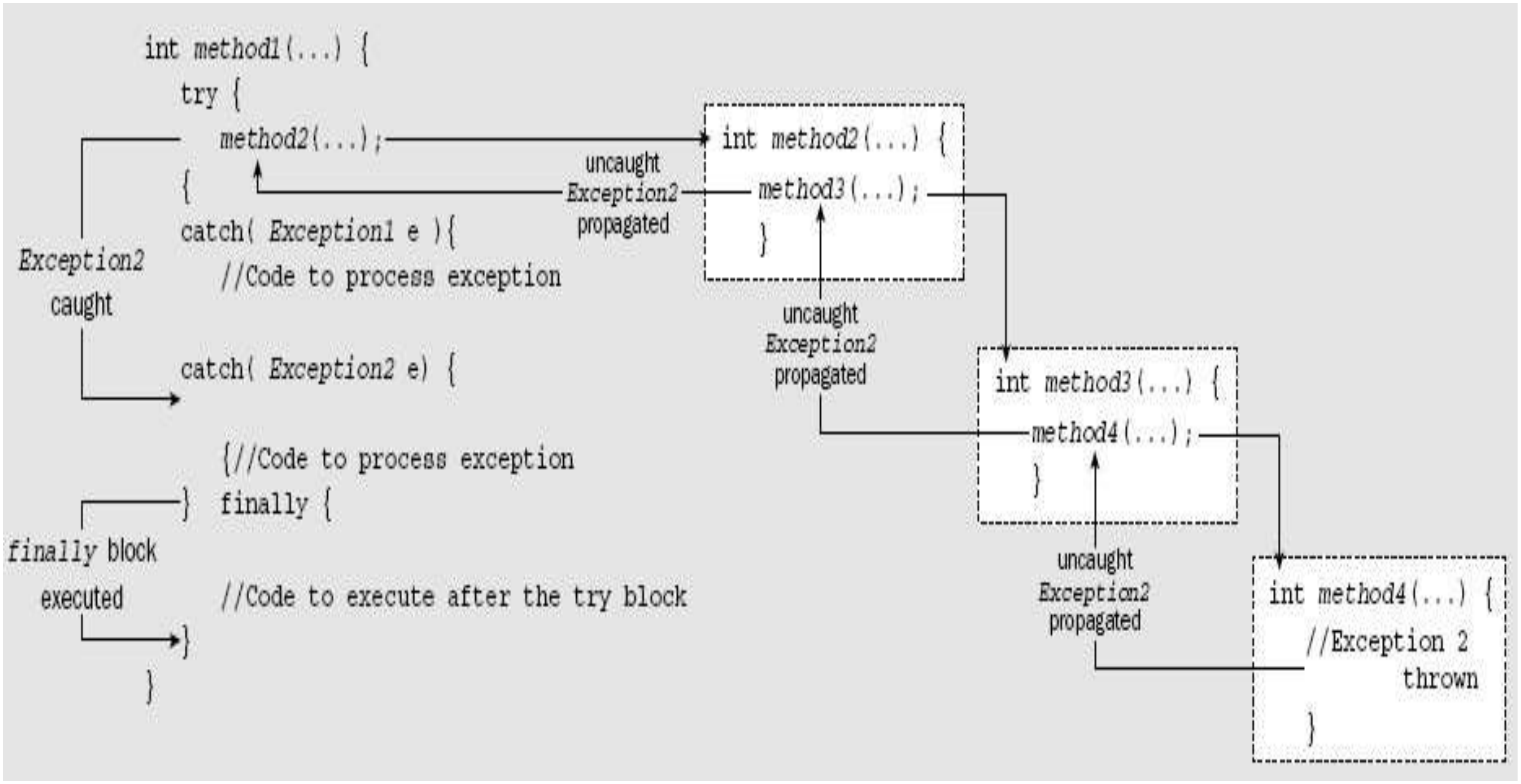
```
try ( ClassName theObject = new ClassName() ) {
   // use theObject here
}catch ( Exception e ){
   // catch exceptions that occur while using the resource
}
```

- You can allocate multiple resources in the parentheses following try by separating them with a semicolon (;).

Call Stack Mechanism

- If an exception is not handled in the current `try-catch` block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

Call Stack Mechanism



```
int method1(...) {
   try {
      method2(...);
   }
   catch( Exception1 e ){
      //Code to process exception

   catch( Exception2 e) {

      {//Code to process exception
   }  finally {

      //Code to execute after the try block
   }
}
```

```
int method2(...) {
   method3(...);
   }
```

```
int method3(...) {
   method4(...);
   }
```

```
int method4(...) {
   //Exception 2
          thrown
   }
```

The Handle or Declare Rule

Use the *handle or declare rule* as follows:

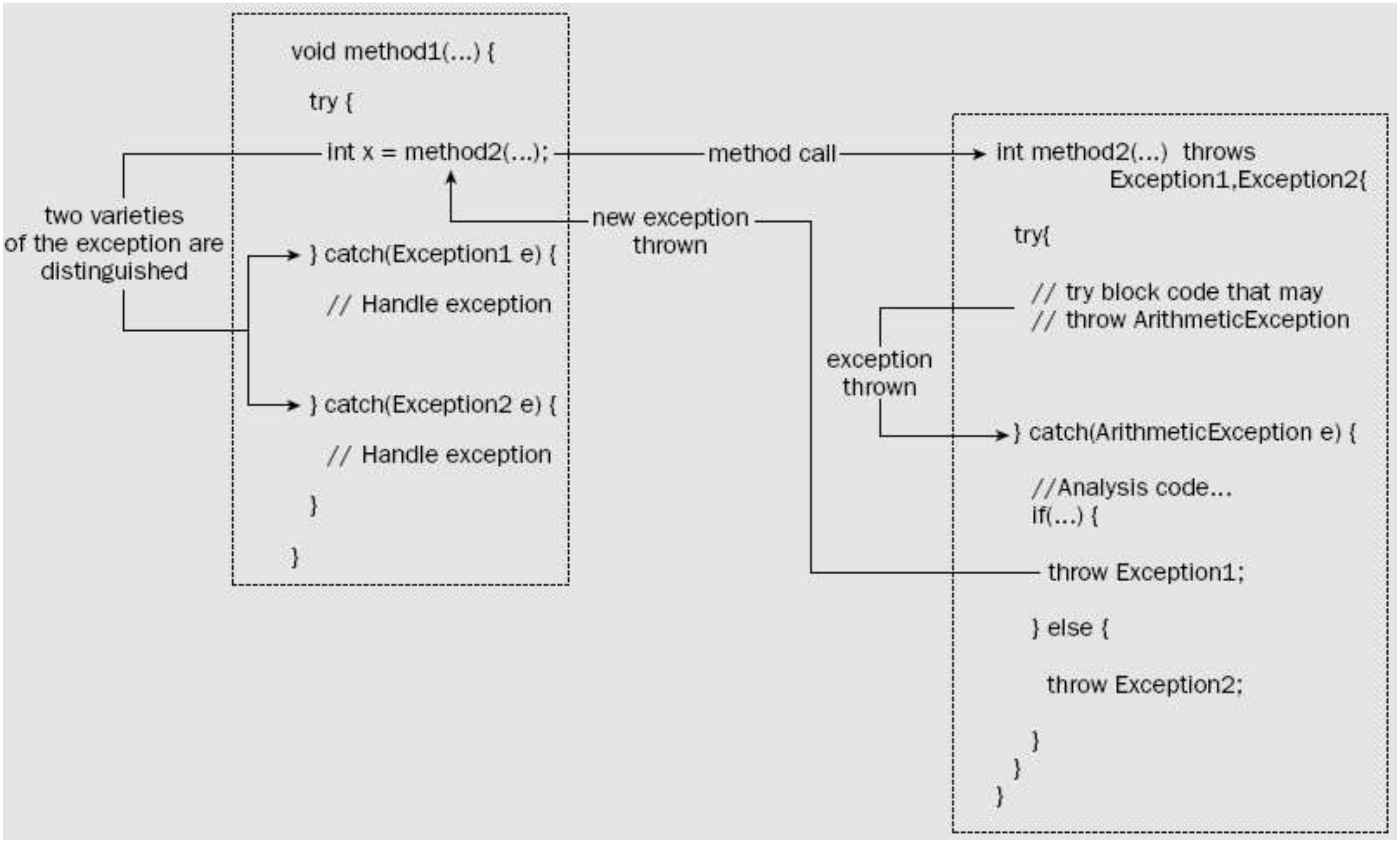
- Handle the exception by using the `try-catch-finally` block.
- Declare that the code causes an exception by using the `throws` clause.

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

Other Principles

- You do not need to declare runtime exceptions or errors.
- You can choose to handle runtime exceptions.

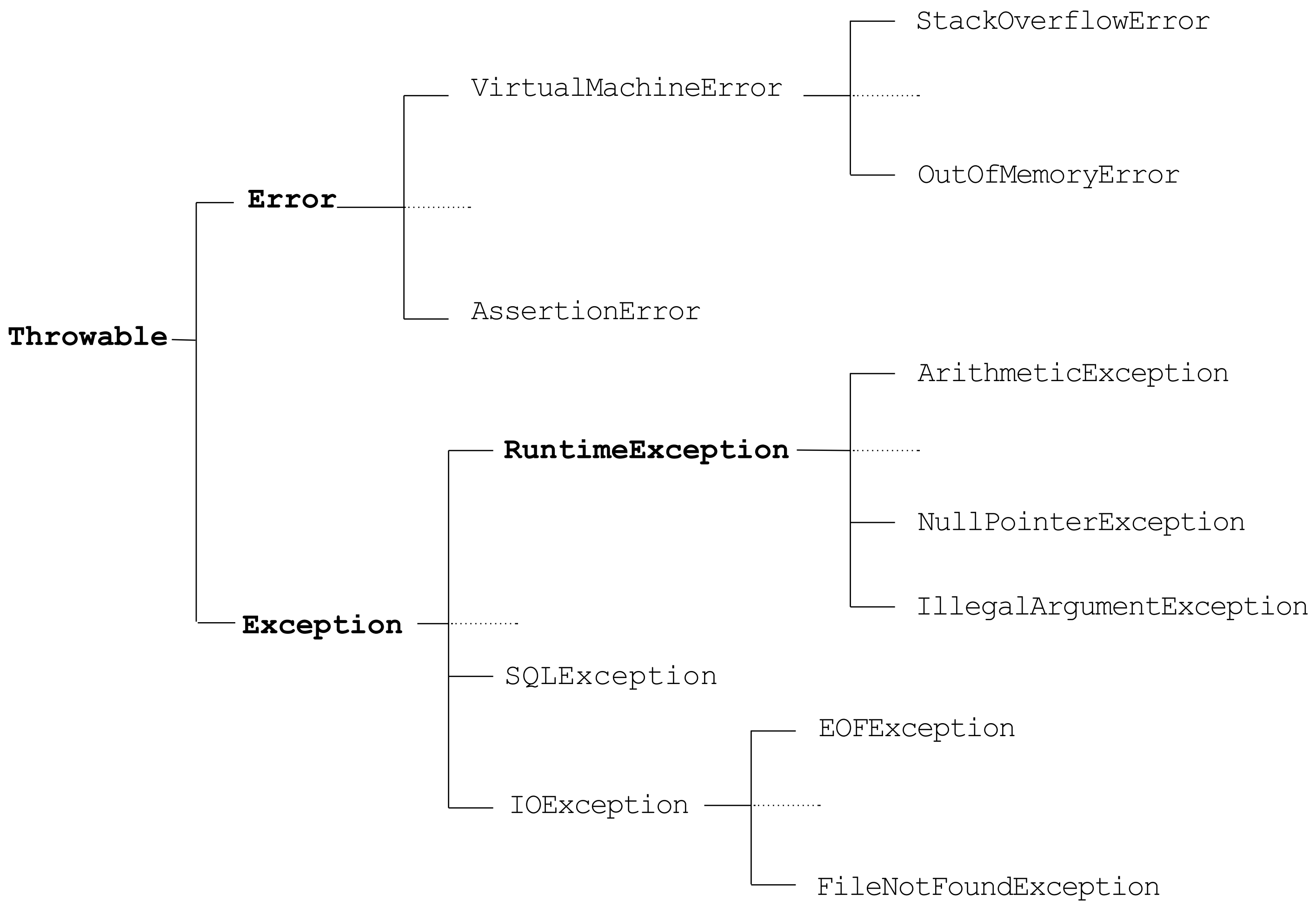
Handle or Declare



```
void method1(...) {
  try {
    int x = method2(...);
  } catch(Exception1 e) {
    // Handle exception
  } catch(Exception2 e) {
    // Handle exception
  }
}
```

```
int method2(...)  throws
              Exception1,Exception2{
  try{
    // try block code that may
    // throw ArithmeticException
  } catch(ArithmeticException e) {
    //Analysis code...
    if(...) {
      throw Exception1;
    } else {
      throw Exception2;
    }
  }
}
```

Exception Categories



Common Exceptions

- `NullPointerException`
- `FileNotFoundException`
- `NumberFormatException`
- `ArithmeticException`
- `SecurityException`

Method Overriding and Exceptions

The overriding method can throw:

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overriding method cannot throw:

- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method

Method Overriding and Exceptions

```
public class TestA {
  public void methodA() throws IOException {
    // do some file manipulation
  }
}
```

```
public class TestB1 extends TestA {
  public void methodA() throws EOFException {
    // do some file manipulation
  }
}
```

```
public class TestB2 extends TestA {
  public void methodA() throws Exception { // WRONG
    // do some file manipulation
  }
}
```

Custom Exception Classes

- **You can declare your own exception classes**

If no existing class meets your needs, think about whether to subclass *Exception* or *RuntimeException* (be checked or unchecked ?).

Name your class with an *Exception* suffix.

public class *InvalidMediaFormatException* extends *Exception*
{...}

- **MyException.java & TestMyException.java**

Creating Your Own Exceptions

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String message, int port) {
    super(message);
    this.port = port;
  }

  public int getPort()
    { return port;
  }
}
```

Use the `getMessage` method, inherited from the `Exception` class, to get the reason for which the exception was made.

Handling a User-Defined Exception

A method can throw a user-defined, checked exception:

```
public void connectMe(String serverName)
       throws ServerTimedOutException
  { boolean successful;
  int portToConnect = 80;

  successful = open(serverName, portToConnect);

  if ( ! successful ) {
    throw new ServerTimedOutException("Could not connect",
                                         portToConnect);
  }
}
```

Handling a User-Defined Exception

Another method can use a `try-catch` block to capture user-defined exceptions:

```
public void findServer()
    { try {
      connectMe(defaultServer);
    } catch (ServerTimedOutException e) {
      System.out.println("Server timed out, trying alternative");
      try {
        connectMe(alternativeServer);
      } catch (ServerTimedOutException e1) {
        System.out.println("Error: " + e1.getMessage() +
                                     " connecting to port " + e1.getPort());
      }
    }
  }
```

Assertions*

Assertions

- Runtime BUGs probably fail unexpectedly when application runs, and the cause of failure can be very difficult to determine.

- Assertions let the developer codify assumptions about application correctness

- When the application runs, and if an assertion fails, the application terminates with a message that helps the developer diagnose the failure’s cause.

Assertions

- Syntax of an assertion is:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

- If `<boolean_expression>` evaluates `false`, then an `AssertionError` is thrown.
- The second argument is converted to a string and used as descriptive text in the `AssertionError` message.

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as if the check was never there.
- Assertion checks are disabled by default. Enable assertions with the following commands:

  ```
  java -enableassertions MyProgram
  ```

  or:

  ```
  java -ea MyProgram
  ```

- Assertion checking can be controlled on class, package, and package hierarchy bases, see: `docs/guide/language/assert.html`

Recommended Uses of Assertions

Use assertions to document and verify the assumptions and internal logic of a single method:

- Internal invariants
- Control flow invariants
- Postconditions and class invariants

Inappropriate Uses of Assertions

- Do not use assertions to check the parameters of a public method.
- Do not use methods in the assertion check that can cause side-effects.

Logs*

Log

- Class ***Logger*** in package ***java.util.logging***
- Logger objects may be obtained by calls ***getLogger()*** factory methods

  – *public static Logger **getLogger**(String **name**)*

  – Create a new Logger or return a suitable existing Logger by the ***name*** of the Logger

  – ***Logger.GLOBAL LOGGER NAME***
- Each Logger has a "*Level*" associated with it.

  – *java.util.logging.**Level** – From highest to lowest: SEVERE, WARNING, INFO, CONFIG, FINE,FINER,FINEST.*
- Log a message with ***log()*** method

  – *void log(Level **level**, String **msg**)*
- TestLogger.java

Deprecation

Deprecation

- Deprecation makes classes, attributes, methods, constructors, and so on, obsolete
- Obsolete declarations are replaced by methods with a more standardized naming convention
- When migrating code, compile the code with the `-deprecation` flag:

```
javac -deprecation MyFile.java
```

Deprecation

A Java 2 SDK version rewritten is:

```
package myutilities;

import java.util.*;
import java.text.*;

public final class DateConverter2
  {  private static String
  DAY_OF_THE_WEEK[] =
      {"Sunday", "Monday", "Tuesday", "Wednesday",

  public static String getDayOfWeek (String theDate) {
    Date d = null;
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yy");

    try {
      d = sdf.parse (theDate);
    } catch (ParseException e)
      { System.out.println (e);
      e.printStackTrace();
    }

    // Create a GregorianCalendar object
    Calendar c =
        new
            GregorianCalendar( TimeZone.getTimeZone
            ("EST"),Locale.US);

    return(
        DAY_OF_THE_WEEK[(c.get(Calendar.DAY_OF_WEEK)-1)]);
  }
}
```

Annotations*

Annotations

- An annotation is an instance of an annotation type and associates metadata with an application element.
- An annotation is expressed in source code by prefixing the type name with the @ symbol.

  @Readonly is an annotation and *Readonly* is its type.

- Annotations can be used to associate metadata with *constructors*, *fields*, *local variables*, *methods*, *packages*, *parameters*, and *types* (annotation, class, enum, and interface).

Annotations

- The compiler supports the *Override*, *Deprecated*, *SuppressWarnings*, *FunctionalInterface*, and *SafeVarargs* annotation types(in the *java.lang* package).
- *@Override* used for expressing a subclass method over-rides a method in the superclass, and does not overload that method instead.

```
@Override
public void draw(int color)
{
   // drawing code
}
```

Annotations

- *@Deprecated* used for indicating the element is deprecated (phased out) and should no longer be used.

```
class Employee{
 /**
 * Employee’s name
 * @deprecated New version uses firstName and lastName fields.
 */
 @Deprecated
 String name;
 String firstName;
 String lastName;
 public static void main(String[]
   args){ Employee emp = new Employee();
   emp.name = “John Doe”;
 }
}
```

Annotations

- *@SuppressWarnings* annotations used for suppressing deprecation or unchecked warnings via a “deprecation” or “unchecked” argument.

```
class UseEmployee{
  @SuppressWarnings(”deprecation”)
  public static void main(String[] args)
                                                    {
   Employee emp = new Employee();
  emp.name = “John Doe”;
  }
}
```

- *@SafeVarargs* annotations used for asserting the body of the annotated method/constructor does not perform potentially unsafe operations on its variable

Annotations

- *@FunctionalInterface*: The annotated type satisfies the requirements of a functional interface.
- A functional interface has exactly one abstract method
  –also as SAM Interface, Single Abstract Method interfaces.
  –instances can be created with lambda expressions, method references, or constructor references.
  –Can include default/static method
- Defining an functional interface:

```
@FunctionalInterface interface GreetingService {
    void sayMessage(String message);
```

- Create an instance with lambda expressions:

```
GreetingService greetSev = message ->
    System.out.println("Hello " + message);
```

- E.g. java.lang.Runnable, java.awt.event.ActionListener

Declaring Your Annotation Types

- Java also lets you declare your own annotation types.
- Uses @interface to declare an annotation type.

```
//@Stub is used to mark empty methods (stubs).
public @interface Stub{
}

public class Deck
  { @Stub
  public void shuffle(){
    // empty method and will be coded later.
  }
}
```

Summary

- Exceptions
- The **`try-catch-finally`** Statement
- Exception Handle Call Stack Mechanism
- Exception Categories
- Method Overriding and Exceptions
- Creating Your Own Exceptions
- Assertions*
- Logs*
- Deprecation*
- Annotations*