infoSupport
Solid Innovator

JAVA Programming

Statements and Exceptions

Overview

- Java Syntax
- Statement Blocks
- Types of Statements
- Handling Basic Exceptions

Java Syntax

- This course does not cover all syntax options.
- For a complete overview search for: “`the java language specification`”
- Java Language Specification can be downloaded

Statement Blocks

Use braces As block delimiters

```
{
  // code
}
```

A block and its parent block cannot have a variable with the same name

```
{
  int i;
  ...
  {
    int i;
    ...
  }
}
```

Sibling blocks can have variables with the same name

```
{
  int i;
  ...
}
...
{
  int i;
  ...
}
```

Types of Statements

Selection Statements

The if and switch statements

Iteration Statements

The while, do, for, and enhanced for

Jump Statements

The break, and continue statements

Selection Statements

- If – then – else statement

```
if(boolean expression){
Statements in case expression evaluates to true;
}
else{
Statements in case expression evaluates to false;
}
```

```
int value = 5;
if (value % 2 == 0) {
        System.out.println("even");
} else {
        System.out.println("odd");
}
```

Selection Statements

- Testing various conditions

```
long sales=123456;
int bonus;
if(sales<100000){
        bonus=250;
}else if(sales>100000){
        bonus=800;
}else{
        bonus=(int)sales/100;
}
```

Selection Statements

- The switch statement
  - Elegant way of handling complex conditions

```
switch (expression) {
case value:
       // Statements
       break; // optional
case value:
       // Statements
       break; // optional
// You can have any number of case statements.
default: // Optional
       // Statements
}
```

Selection Statements

```
enum CompassPoint {
        NORTH, EAST, SOUTH, WEST
};
```

```
CompassPoint compassPoint = CompassPoint.WEST;
switch (compassPoint) {
case NORTH:
        System.out.println("Heading north.");
        break;
case EAST:
        System.out.println("Heading east.");
        break;
case SOUTH:
        System.out.println("Heading south.");
        break;
case WEST:
        System.out.println("Heading west.");
        break;
default:
        System.out.println("invalid direction.");
}
```

Iteration Statements

- The while Statement
- The do Statement
- The for Statement
- The enhanced-for Statement (**foreach**)

The while Statement

- Execute embedded statements based on Boolean value
- Evaluate Boolean expression at beginning of loop
- Execute embedded statements while Boolean value Is True

```
while (condition) {
        //statements
}
```

```
int i = 0;
while (i < 10) {
    System.out.printf("%d ",i);
    i++;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

The do Statement

- Execute embedded statements based on boolean value
- Evaluate boolean expression at end of loop
- Execute embedded statements while boolean value is true

```
do {
    //statements
} while (condition);
```

```
int i = 0;
do {
    System.out.printf("%d ",i);
    i++;
} while (i < 10);
```

```
0 1 2 3 4 5 6 7 8 9
```

The for statement

- Compact way to iterate over a range of values
- Repeatedly loops until condition is satisfied

```
for (initialization; condition; increment or decrement)
{
        //statements
}
```

```
for (int i = 0; i < 10; i++) {
        System.out.printf("%d ",i);
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Enhanced for statement

- Designed for iteration through collections
- Also referred at as foreach
- Only looping forwards through collection
- No modifications to items.

```
for (item: collection) {
        //statements
}
```

```
int[] values={0,1,2,3,4,5,6,7,8,9};
for (int value : values) {
        System.out.printf("%d ",value);
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Branching Statements

- Break terminates the (inner most) for, while, do or switch loop

```
int[] values={12,23,34,45,56,67,78,89};
int searchFor=76;
String result="Not Found";
for (int value:values) {
        if(value==searchFor){
                result="Found.";
                break;
        }
}
System.out.println(result);
```

Branching Statements

- A labelled Break exists which terminates the outer for, while, do or switch loop

```
int[ ][ ] values = { { 12, 23, 34 },
                     { 45, 56, 67 },
                     { 78, 89, 90 } };
int searchFor = 67;
String result = "Not Found";
loopEnd:
for (int j = 0; j < values.length; j++) {
        for (int value : values[j]) {
                if (value == searchFor) {
                        result = "Found.";
                        break loopEnd;
                }
        }
}
System.out.println(result);
```

Branching Statements

- Skips the current iteration of a for, while , or do-while loop

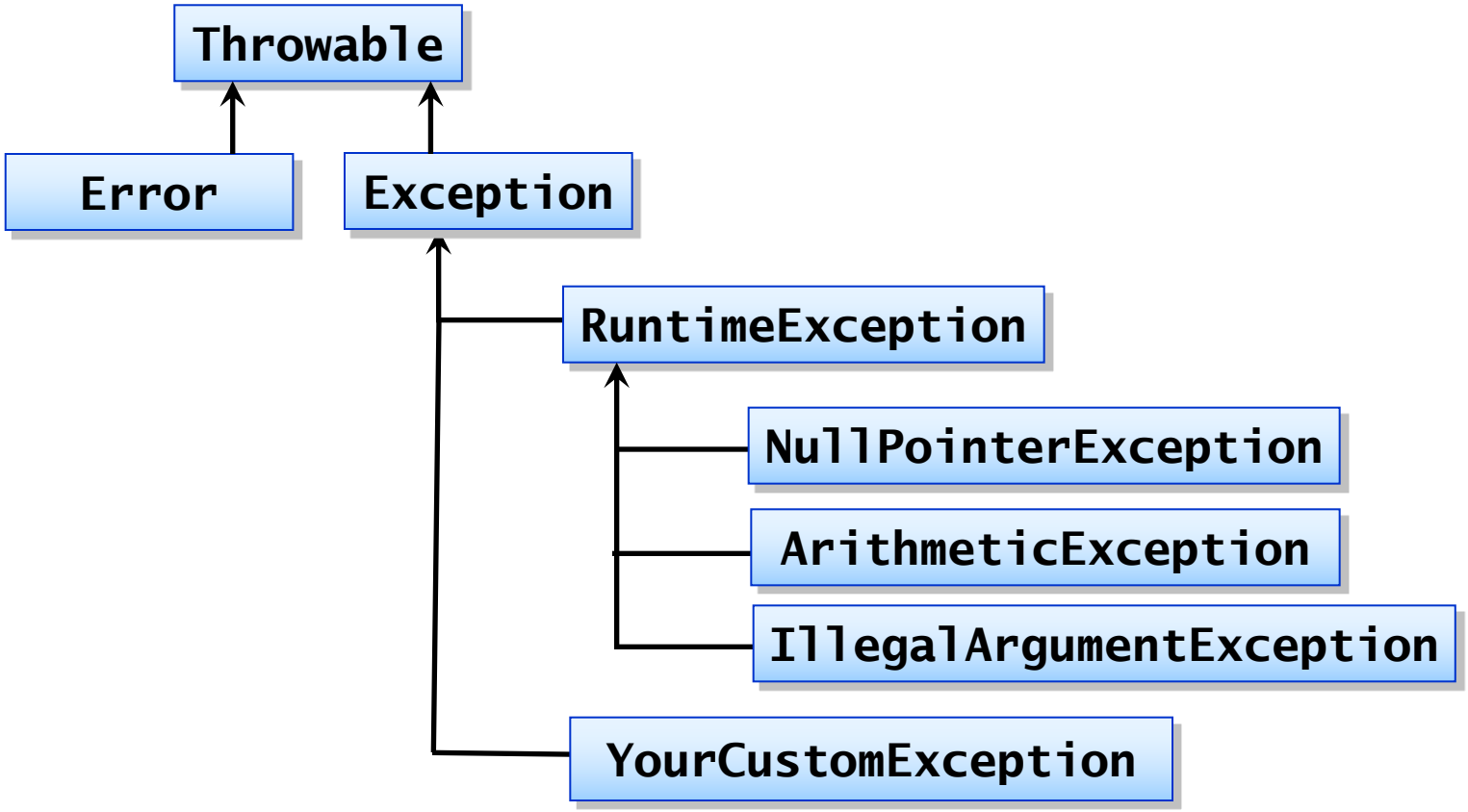
```
int[] values = { 12, 23, 34, 45, 56, 67, 78, 89, 90 };
for (int i = 1; i < 100; i++) {
        for (int j = 0; j < values.length; j++) {
                if (values[j] == i) {
                        System.out.printf("found %d at %d", i, j);
                        System.out.println();
                        continue;
                }
        }
}
```

- A labelled continue exists

Handling Basic Exceptions

- Exception Objects
- Using try and catch Blocks
- Multiple catch Blocks

Exception Objects



Catch or Specify Requirement

- Code that might throw a certain exception must be enclosed by
  - A try / catch block that catches the exception

    or
  - A method that specifies that it can throw the exception.

Kinds of Exceptions

- Error
  - Exceptional conditions external to the application like system or hardware malfunctions.
  - Not subject to Catch or Specify Requirement
- Checked Exceptions
  - Exceptional conditions that a program should recover from
  - Subject to Catch or Specify Requirement.
- Runtime Exception
  - Exceptional conditions that a program can not anticipate or recover from, like logic errors
  - Not subject to Catch or Specify Requirement

Using try and catch Blocks

- Object-oriented solution to exception handling
  - Put the normal code in a **try** block
  - Handle the exceptions in a separate **catch** block

```
try {
        statements                    ← Program logic
}catch (ArithmeticException caught){
        statements                    ← Error handling
}
```

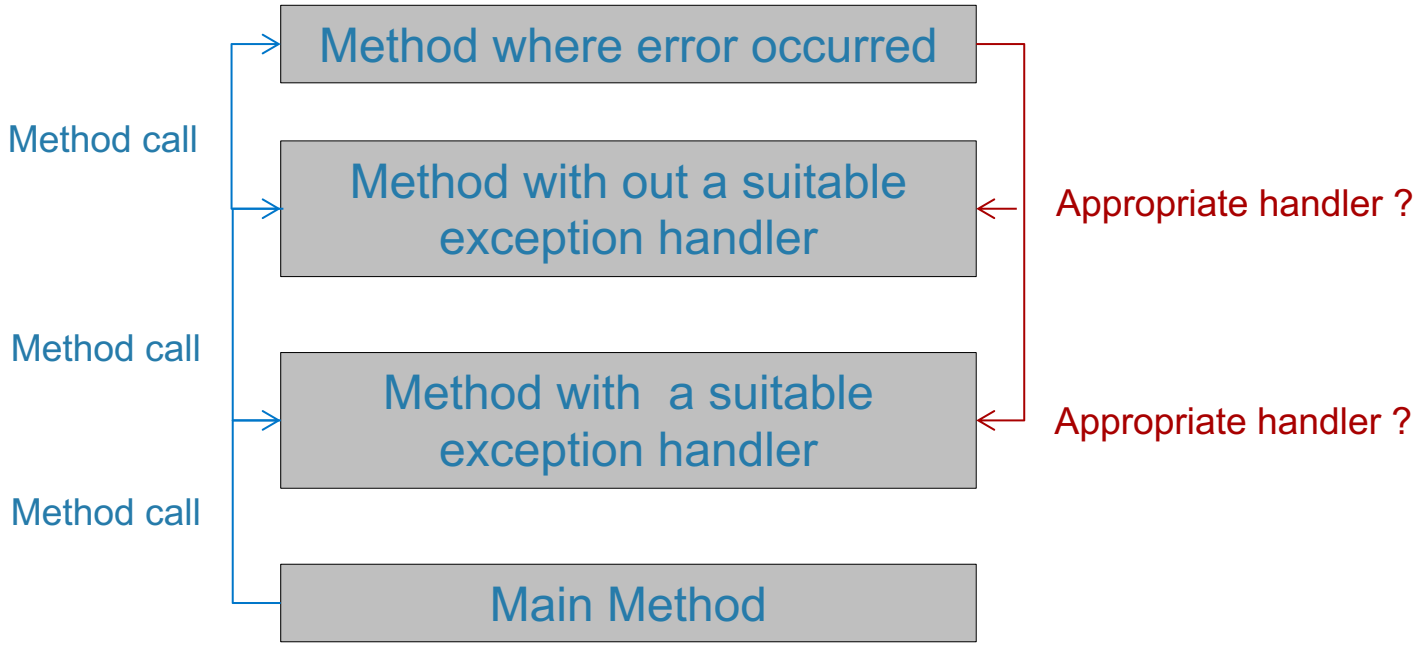
Multiple catch Blocks

- A try block is not allowed to catch a class that is derived from a class caught in an earlier catch block

```
try{
        statements
}catch (exception1 identifier1) {
        statements
}catch (exception2 identifier2) {
        statements
        //Exception2 may not be more specific than exception 1
}catch (exception3 | exception4 identifier3) {
        statements
        //One exception handler for multiple exception types
}finally{
        statements
        //Always executes on exit of try block
}
```

Call stack

- In case of an exception the call stack is traversed to find a suitable handler.



Raising Exceptions

- The throw Statement
- The finally Clause
- Checking for Arithmetic Overflow
- Guidelines for Handling Exceptions

Exceptions thrown by Method

- Specify the exceptions thrown by a method in a throws clause

```
public void methodA() throws IOException, CustomException
```

- Only exceptions not caught in methodA() must be listed.

The throw Statement

- Throw an appropriate exception
- Give the exception a meaningful message

```
try{
        //statements
        throw new CustomException("Custom message")
        //statements

}catch(CustomException ex){
        //statements
        }
```

Throwing an Exception

- Catch a system Exception and throw your own Exception

```
try{
        //statements

}catch(IOException ex){
        //statements
        throw new CustomException(“Custom message”)
        }
```

Guidelines for Handling Exceptions

- Throwing
  - Arithmetic overflow or underflow will never throw an exception
  - Avoid exceptions for normal or expected cases
  - Never create and throw objects of class **Exception**
  - Include a description string in an **Exception** object
  - Throw objects of the most specific class possible
- Catching
  - Arrange **catch** blocks from specific to general
  - Do not let exceptions drop off **main**
- Finally
  - All code in finally block are always executed

Assertions

- Assertion is used to check condition that always should be true
- If assertion returns false an exception is thrown
- Assertion evaluation is off by default
  - Assertion evaluation can be turned on/off when starting the JVM
    - -enableAssertions (-ea)
    - -disableAssertions (-da)
    - In IDE, add VM argument

Assertions

```
public static Person getPerson(int id){
        Person result=null;
        result=persons.get(id);
        if(result==null){
          result=new Person();
        }
        assert (result!=null):"result should not be null";
        return result;
}
```

Assertions

- Use Exceptions to address problems that might occur.
- Use Assertions when checking pre-conditions / post-conditions.
  - Assertions are disabled by default so use them mainly for debugging purposes.

Lab 4: StatementsAndExceptions