

# CMSC 132: Object-Oriented Programming II

---



## Exceptions & Effective Java

Department of Computer Science  
University of Maryland, College Park

1

## Overview

- **Exceptions**
  - Motivation
  - Representation in Java
- **Effective Java**

2

## Exception Handling

- Performing action in response to exception
- Example actions
  - Ignore exception
  - Print error message
  - Request new data
  - Retry action
- Approaches
  1. Exit program
  2. Exit method returning error code
  3. Throw exception

3

## Problem

- May not be able to handle error locally
  - Not enough information in method / class
  - Need more information to decide action
- Handle exception in calling function(s) instead
  - Decide at application level (instead of library)
  - Examples
    - Incorrect data format ⇒ ask user to reenter data
    - Unable to open file ⇒ ask user for new filename
    - Insufficient disk space ⇒ ask user to delete files
- Will need to **propagate** exception to caller(s)

4

## Exception Handling – Exit Program

### ■ Approach

- Exit program with error message / error code

### ■ Example

```
if (error) {  
    System.err.println("Error found");    // message  
    System.exit(1);                       // error code  
}
```

### ■ Problem

- Drastic solution
- Event must be handled by user invoking program
- Program may be able to deal with some exceptions

5

## Exception Handling – Error Code

### ■ Approach

- Exit function with return value ⇒ error code

### ■ Example

```
A() { if (error) return (-1); }  
B() { if ((retval = A()) == -1) return (-1); }
```

### ■ Problems

- Calling function must check & process error code
  - May forget to handle error code
  - May need to return error code to caller
- Agreement needed on meaning of error code
- Error handling code mixed with normal code

6

## Exception Handling – Throw Exception

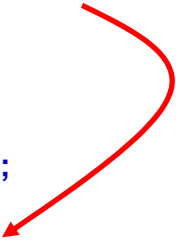
### ■ Approach

- Throw exception (caught in parent's **catch** block)

### ■ Example

```
A() {  
    if (error) throw new ExceptionType();  
}  
B() {  
    try {  
        A();  
    }  
    catch (ExceptionType e) { ...action... }  
}
```

Java exception backtracks to caller(s) until matching catch block found



7

## Exception Handling – Throw Exception

### ■ Advantages

- Compiler ensures exceptions are caught eventually
- No need to explicitly **propagate** exception to caller
  - **Backtrack** to caller(s) automatically
- Class hierarchy defines meaning of exceptions
  - No need for separate definition of error codes
- Exception handling code separate & clearly marked

8

## Representing Exceptions

### ■ Exceptions represented as

- Objects derived from class `Throwable`

### ■ Code

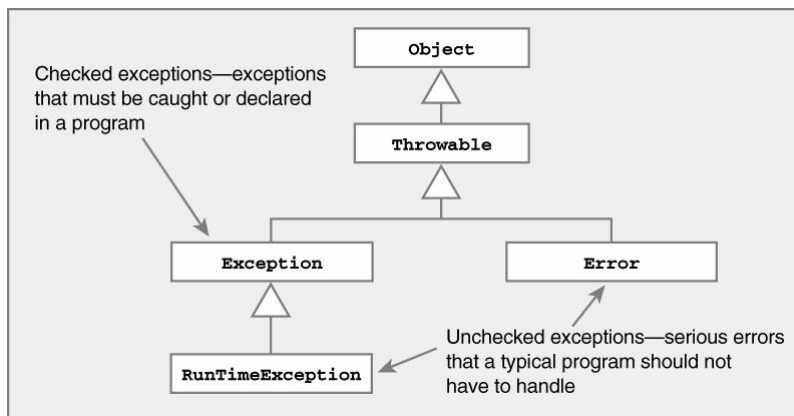
```
public class Throwable( ) extends Object {  
    Throwable( )                // No error message  
    Throwable( String msg )    // Error message  
    String getMessage()        // Return error msg  
    void printStackTrace( ) { ... } // Record methods  
    ...                        // called & location  
}
```

9

## Representing Exceptions

### ■ Java Exception class hierarchy

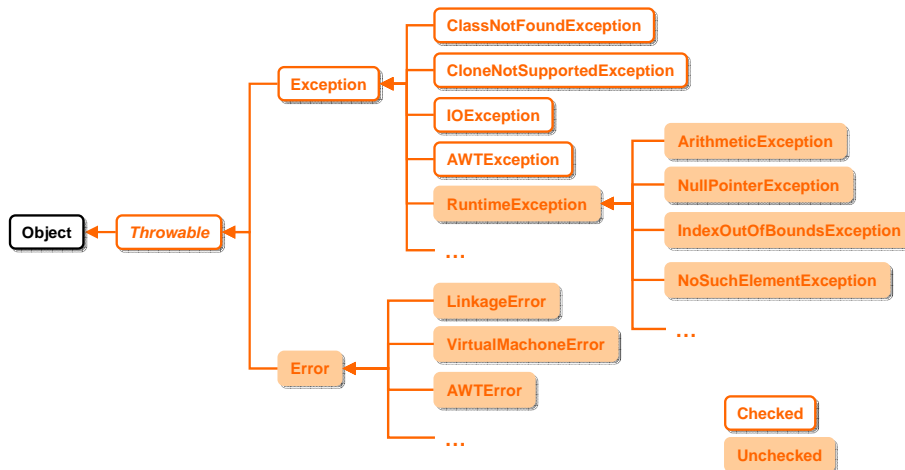
- Two types of exceptions ⇒ checked & unchecked



10

## Representing Exceptions

### ■ Java Exception class hierarchy



11

## Unchecked Exceptions

- Class **Error** & **RunTimeException**
- Serious errors not handled by typical program
- Usually indicate logic errors
- Example
  - **NullPointerException, IndexOutOfBoundsException**
- Catching unchecked exceptions is **optional**
- Handled by Java Virtual Machine if not caught

12

## Checked Exceptions

- Class **Exception** (except RuntimeException)
- Errors typical program should handle
- Used for operations prone to error
- Example
  - IOException, ClassNotFoundException
- Compiler requires “**catch or declare**”
  - Catch and handle exception in method, OR
  - Declare method can throw exception, force calling function to catch or declare exception in turn
  - Example
    - void A( ) throws ExceptionType { ... }

13

## Designing & Using Exceptions

- Use exceptions only for rare events
  - Not for common cases ⇒ checking end of loop
  - High overhead to perform catch
- Place statements that jointly accomplish task into single try / catch block
- Use existing Java Exceptions if possible

14

## Designing & Using Exceptions

- **Avoid simply catching & ignoring exceptions**

- **Poor software development style**

- **Example**

```
try {  
    throw new ExceptionType1();  
    throw new ExceptionType2();  
    throw new ExceptionType3();  
}  
catch (Exception e) { // catches all exceptions  
    ...                // ignores exception & returns  
}
```

15

## Overview

- **Exceptions**

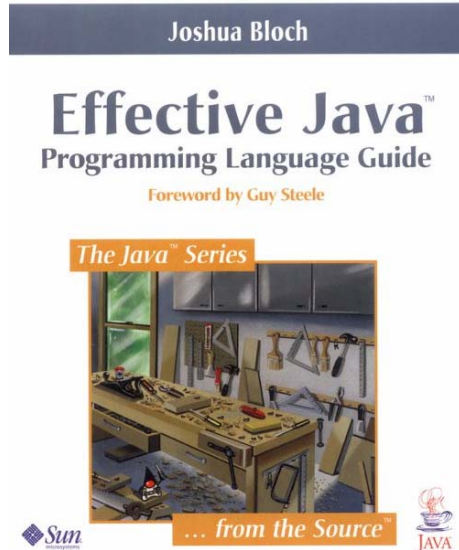
- **Effective Java**

- **Puzzlers**
- **Principles**

16

## Effective Java

- **Title**
  - Effective Java Programming Language Guide
- **Author**
  - Joshua Bloch
- **Contents**
  - Useful tips for Java programming



17

## Java Puzzlers (By J. Bloch)

- **Java**
  - Simple and elegant
  - Need to avoid some sharp corners!
- **Puzzlers**
  - Java code fragments
  - Expose some tricky aspects of Java
- **Effective Java**
  - Ways of avoiding Java programming problems

18

## What's In A Name?

```
public class Name {
    private String myName;
    public Name(String n) { myName = n; }
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return myName.equals(n.myName);
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Donald"));
        System.out.println(
            s.contains(new Name("Donald")));
    }
}
```

### Output

1. True
2. False
3. It Varies

**Name class violates Java hashCode( ) contract.**

**If you override equals( ), must also override hashCode( )!**

19

## You're Such A Character

```
public class Trivial {
    public static void main(String args[] ) {
        System.out.print("H" + "a");
        System.out.print('H' + 'a');
    }
}
```

### Output

1. Ha
2. HaHa
3. Neither

**Prints Ha169**

**'H' + 'a' evaluated as *int*, then converted to String!**

**Use string concatenation (+) with care. At least one operand must be a String**

20

## The Confusing Constructor

```
public class Confusing {
    public Confusing(Object o) {
        System.out.println("Object");
    }
    public Confusing(double[] dArray) {
        System.out.println("double array");
    }
    public static void main(String args[]) {
        new Confusing(null);
    }
}
```

### Output

1. Object
2. double array
3. Neither

When multiple overloads apply, the most specific wins

Avoid overloading. If you overload, avoid ambiguity

21

## Time For A Change

### ■ Problem

- If you pay \$2.00 for a gasket that costs \$1.10, how much change do you get?

```
public class Change {
    public static void main(String args[] ) {
        System.out.println(2.00 - 1.10);
    }
}
```

### Output

1. 0.9
2. 0.90
3. Neither

Prints 0.8999999999999999. Decimal values can't be represented exactly by float or double

Avoid float or double where exact answers are required. Use BigDecimal, int, or long instead

22

## A Private Matter

```
class Base {
    public String name = "Base";
}
class Derived extends Base {
    private String name = "Derived";
}
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived( ).name);
    }
}
```

### Output

1. Derived
2. Base
3. Neither

Compiler error  
in class  
PrivateMatter:  
Can't access  
name

Private field  
can hide public.  
Avoid hiding &  
public fields

23

## Effective Java Topics

1. Creating and Destroying Objects
2. Methods Common to All Objects
3. Classes and Interfaces
4. Substitutes for C Constructs
5. Methods
6. General Programming
7. Exceptions
8. Threads
9. Serialization

24

## Creating and Destroying Objects

- Consider providing static factory methods instead of constructors
- Enforce singleton property with a private constructor
- Enforce noninstantiability with a private constructor
- Avoid creating duplicate objects
- Eliminate obsolete object references
- Avoid finalizers

25

## Methods Common to All Objects

- Obey the general contract when overriding equals
- Always override hashCode when you override equals
- Always override toString
- Override clone judiciously
- Consider implementing Comparable

26

## Classes and Interfaces

- Minimize the accessibility of classes and members
- Favor immutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it
- Prefer interfaces to abstract classes
- Use interfaces only to define types
- Favor static member classes over nonstatic

27

## Methods

- Check parameters for validity
- Make defensive copies when needed
- Design method signatures carefully
- Use overloading judiciously
- Return zero-length arrays, not nulls
- Write doc comments for all exposed API elements

28

## General Programming

- Minimize the scope of local variables
- Know and use the libraries
- Avoid float and double if exact answers are required
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Prefer interfaces to reflection
- Use native methods judiciously
- Optimize judiciously
- Adhere to generally accepted naming conventions

29

## Exceptions

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and run-time exceptions for programming errors
- Avoid unnecessary use of checked exceptions
- Favor the use of standard exceptions
- Throw exceptions appropriate to the abstraction
- Document all exceptions thrown by each method
- Include failure-capture information in detail messages
- Strive for failure atomicity
- Don't ignore exceptions

30

## Threads

- Synchronize access to shared mutable data
- Avoid excessive synchronization
- Never invoke wait outside a loop
- Don't depend on the thread scheduler
- Document thread safety
- Avoid thread groups