Java Programming Basics (II)

Desenvolvimento de Software e Sistemas Móveis (DSSMV)

Licenciatura em Engenharia de Telecomunicações e Informática

LETI/ISEP

2025/26

Paulo Baltarejo Sousa

pbs@isep.ipp.pt

Instituto Superior de Engenharia do Porto

P.PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

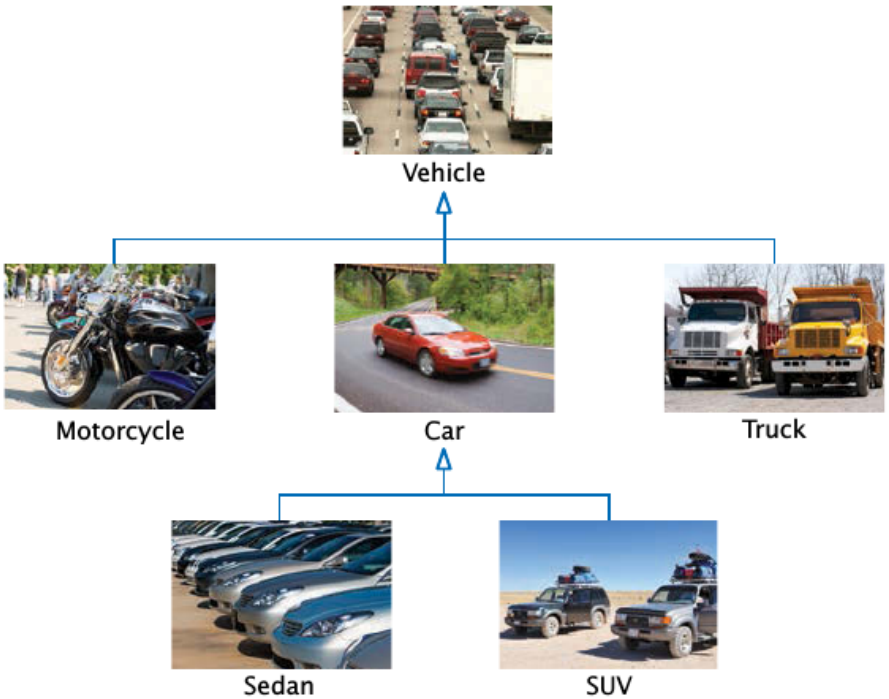
- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

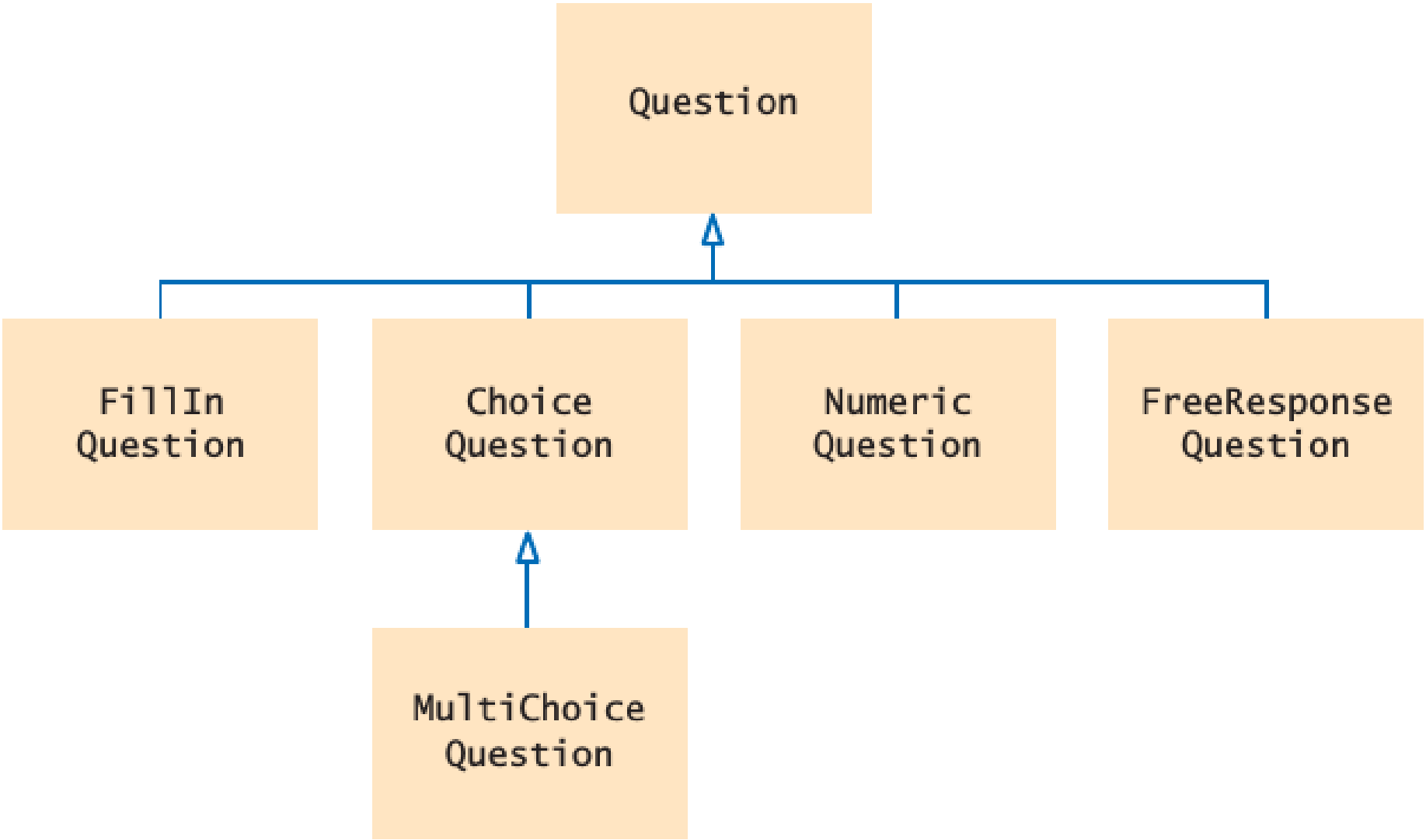
Inheritance & Polymorphism

Deriving a class

- In object-oriented design, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**).
- The subclass inherits data and behavior from the superclass.



Example


Question
FillIn Question
Choice Question
Numeric Question
FreeResponse Question
MultiChoice Question

Class `Question`

```
package com.company;

public class Question {
   private String text;
   private String answer;

   public Question(){
      text = "";
      answer = "";
   }
   public void setText(String questionText){
      text = questionText;
   }
   public void setAnswer(String correctResponse){
      answer = correctResponse;
   }
   public boolean checkAnswer(String response){
      return response.equals(answer);
   }
   public void display(){
      System.out.println(text);
   }
}
```

Testing class `Question`

```
package com.company;

import java.util.Scanner;

public class Main {

   public static void main(String[] args) {
      Question q = new Question();
      q.setText("Who was the inventor of Java?");
      q.setAnswer("James Gosling");
      presentQuestion(q);
   }
   public static void presentQuestion(Question q){
      q.display();
      System.out.print("Your answer: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
   }
}
```

Implementing subclass

- The `extends` reserved word indicates that a class inherits from a superclass.
  - A subclass can **override** a superclass method by providing a new implementation.
  - **A subclass inherits all methods** that it does not override.
- Use the reserved word `super` to call a superclass method or constructor.

class ChoiceQuestion

```
public class ChoiceQuestion extends Question {
   private ArrayList<String> choices;
   public ChoiceQuestion() {
      choices = new ArrayList<String>();
   }
   public void addChoice(String choice, boolean correct) {
      choices.add(choice);
      if (correct) {
         String choiceString = "" + choices.size();
         setAnswer(choiceString);//invoking Question setAnswer method
      }
   }
   @Override
   public void display() {
      super.display(); //invoking Question display method
      for (int i = 0; i < choices.size(); i++) {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

- Why `Question` class `display` method is invoked using `super` and `Question` class `setAnswer` method not?

Testing class `ChoiceQuestion`

```
public class Main{
   public static void main(String[] args) {
      ChoiceQuestion first = new ChoiceQuestion();
      first.setText("What was the original name of the Java language?");
      first.addChoice("*7", false);
      first.addChoice("Duke", false);
      first.addChoice("Oak", true);
      first.addChoice("Gosling", false);

      ChoiceQuestion second = new ChoiceQuestion();
      second.setText("In which country was the inventor of Java born?");
      second.addChoice("Australia", false);
      second.addChoice("Canada", true);
      second.addChoice("Denmark", false);
      second.addChoice("United States", false);

      presentQuestion(first);
      presentQuestion(second);
  }
  public static void presentQuestion(Question q){
      q.display();
      System.out.print("Your answer: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
}
```

Calling the Superclass Constructor

```
public class Question {
   private String text;
   private String answer;

   public Question() {
      this.text = "";
      answer = "";
   }
   public Question(String text) {
      this.text = text;
      answer = "";
   }
   ...
}
```

```
public class ChoiceQuestion extends Question {
   private ArrayList<String> choices;
   ...
   public ChoiceQuestion(String text){
      super(text);
      choices = new ArrayList<String>();
   }
   ...
}
```

- Unless specified otherwise, the **subclass constructor calls the superclass constructor with no arguments**.
- To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.
  - The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Polymorphism

- Polymorphism ("having multiple forms) allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
- In order to present a question to the user, we **need not know** the exact type of the question.
  - We just display the question and check whether the user supplied the correct answer.
  - The `Question` superclass has methods for displaying and checking. Therefore, we can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`

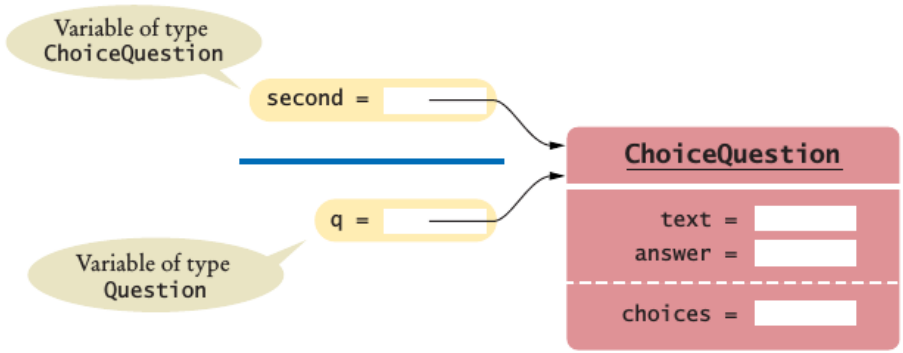
```
public static void presentQuestion(Question q){
   q.display();
   System.out.print("Your answer: ");
   Scanner in = new Scanner(System.in);
   String response = in.nextLine();
   System.out.println(q.checkAnswer(response));
}
```

Dynamic Method Lookup (I)

- A **subclass reference** can be used when a **superclass reference** is expected.

```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

- When the `presentQuestion` method executes, the object references stored in `second` and `q` refer to the same object of type `ChoiceQuestion`.



Dynamic Method Lookup (II)

```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

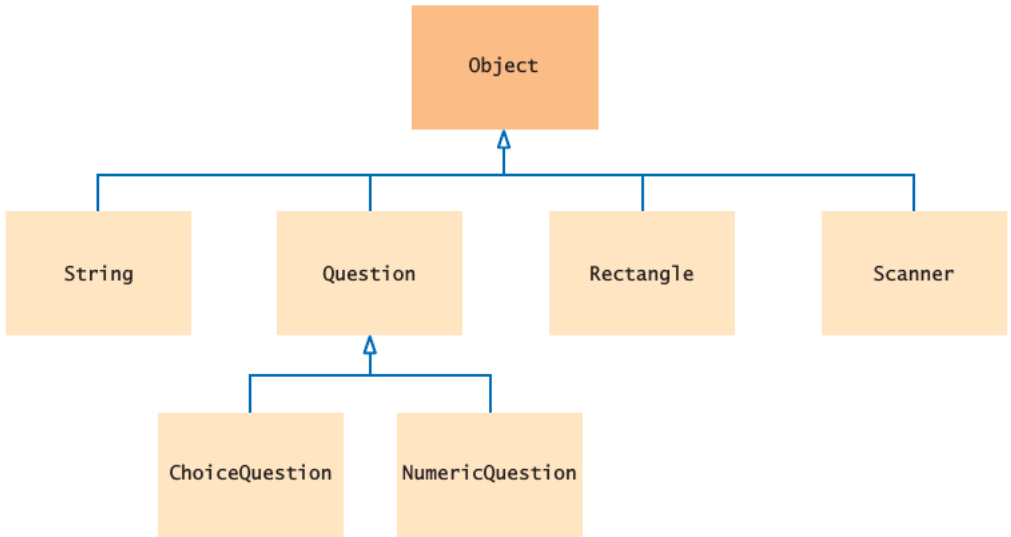
```
public static void presentQuestion(Question q){
   q.display();
   ...
}
```

- `q.display();`
  - Does it call `Question.display()` or `ChoiceQuestion.display()`?
- In Java, method calls **are always determined by the type of the actual object**, not the type of the variable containing the object reference.
  - This is called **dynamic method lookup**.

Object: The Cosmic Superclass

Class Object

- In Java, every class that is declared without an explicit `extends` clause automatically extends the class `Object`.
  - **The class `Object` is the direct or indirect superclass of every class in Java**.
  - The `Object` class defines several very general methods, including
    - `toString`, which yields a string describing the object.
    - `equals`, which compares objects with each other.
    - `hashCode`, which yields a numerical code for storing the object in a set.



Upcasting & Downcasting

- **Upcasting is casting to a supertype**, while **downcasting is casting to a subtype**.
- Upcasting **is always allowed**.
  - It is correct to **store a subclass reference** in a **superclass variable**

```
ChoiceQuestion cq = new ChoiceQuestion();
Question q = cq; // OK
Object anObject = cq; // OK
```

- Downcasting involves a type check (requires the `instanceof` operator) and can throw a `ClassCastException`.
  - **store a superclass reference** in a **subclass variable**

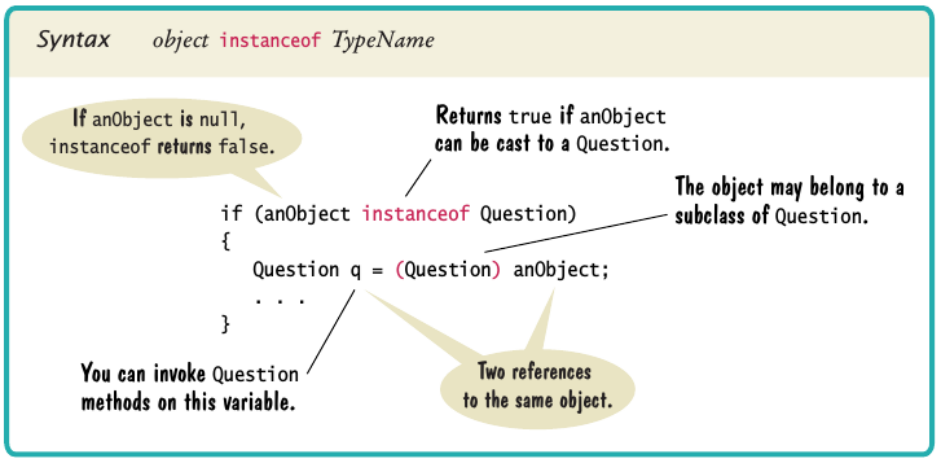
The instanceof Operator

- **To protect against bad casts**, you can use the `instanceof` operator.
  - **It tests whether an object belongs to a particular type**.

```
Object anObject = new Question();

if(anObject instanceof Question){
    Question q = (Question) anObject;
}
```

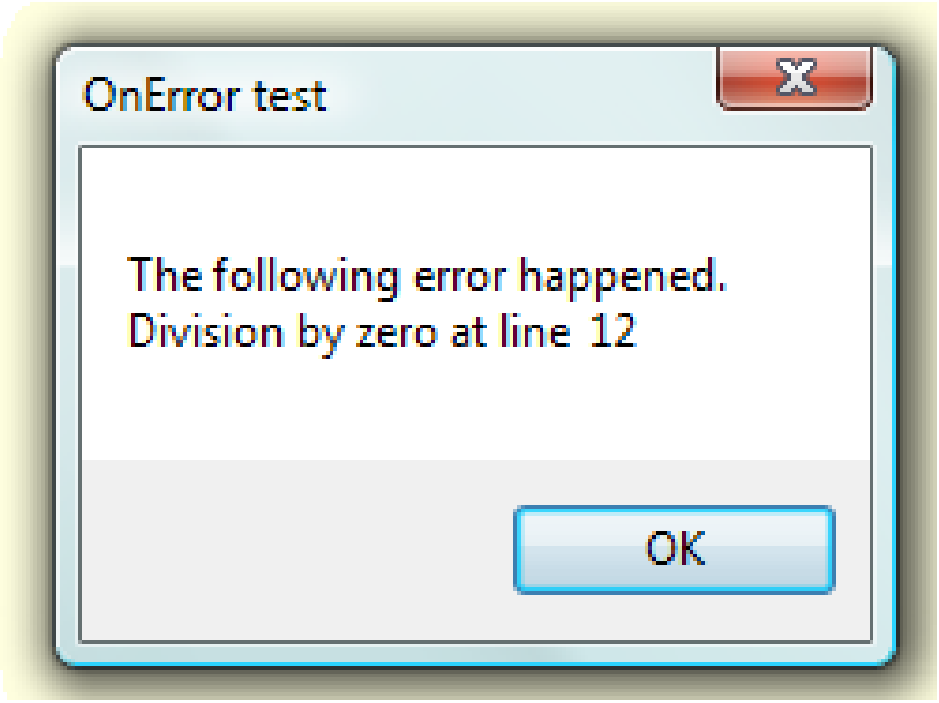
  - It returns `true` if the type of `anObject` is convertible to `Question`.



Exceptions

Overview

- There are two aspects to dealing with program errors: **detection** and **handling**.

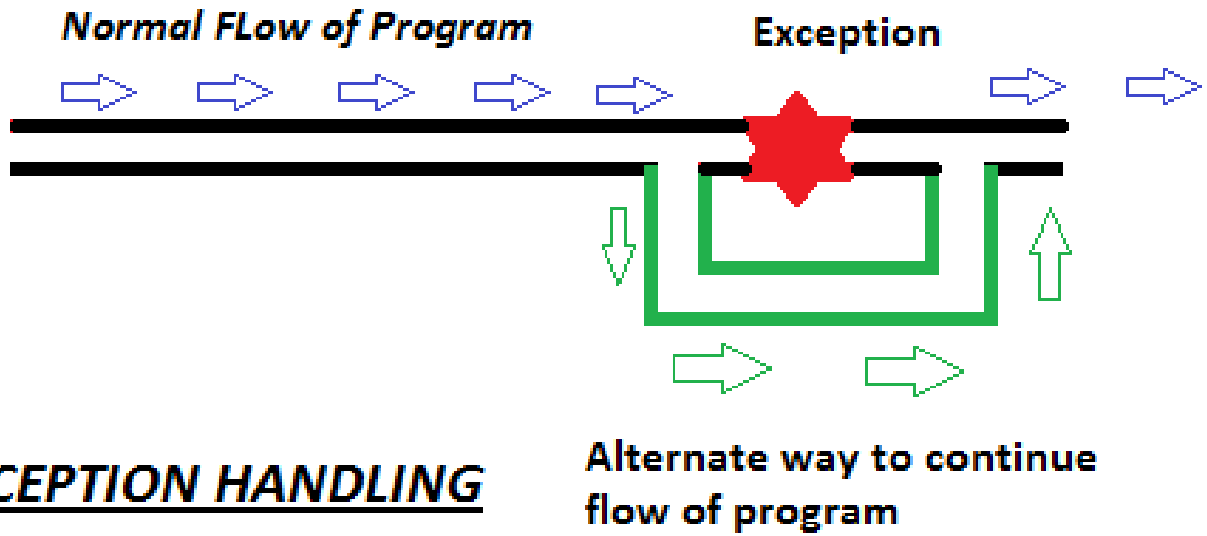


Exception

- An **exception** is an unexpected or error condition.
  - A program might issue a command to read a file from a disk, but the file does not exist there.
  - A program might attempt to write data to a disk, but the disk is full or unformatted.
  - A program might ask for user input, but the user enters an invalid data type.
  - A program might attempt to divide a value by 0.
  - A program might try to access an array with a subscript that is too large or too small.

Exception Handling

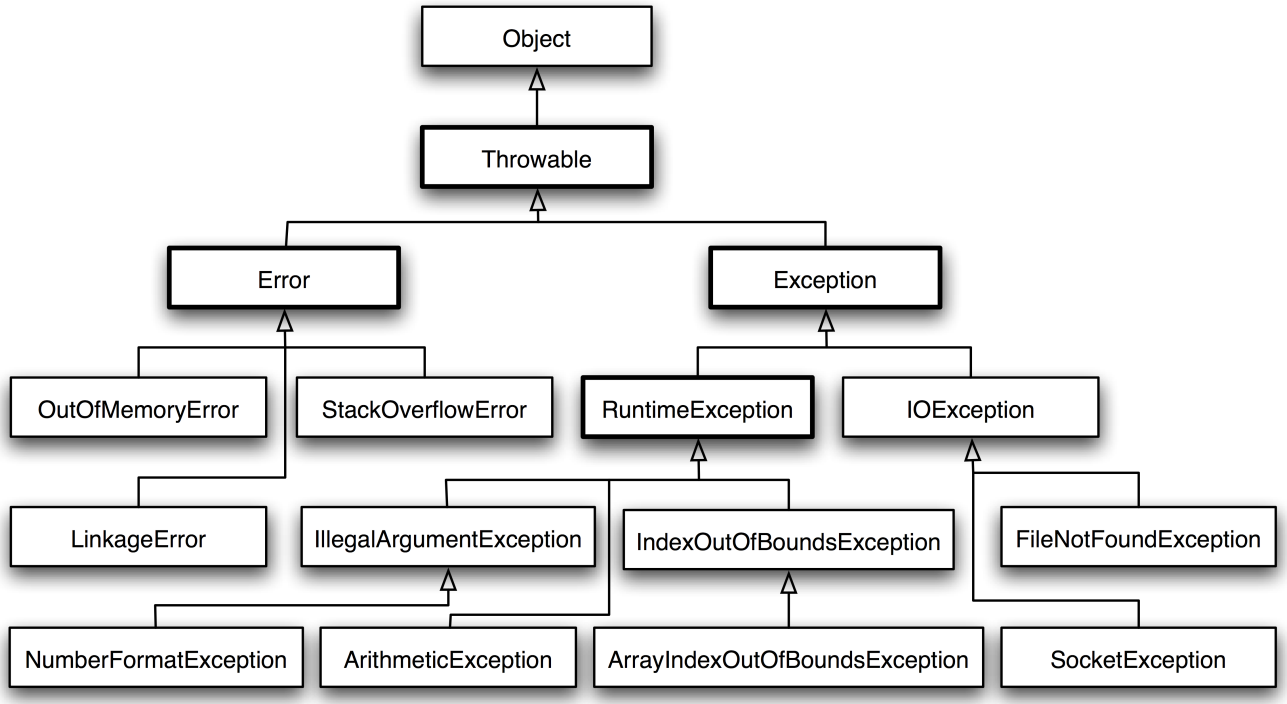
- It provides a flexible mechanism **for passing control from the point of error detection to a handler** that can deal with the error.



- When **an exception is throwable**, **execution does not continue with the next statement**, but with an **exception handler**.

Class `Exception`

- **When an exceptional condition arises, an object representing such exception is created.**
  - The exception is **caught and processed**.



Handling Exceptions (I)

- Java exception handling is managed via five keywords: `try`, `catch`, `throw`, `throws`, and `finally`.
  - Aplication statements that you want to **monitor (detect)** for exceptions are contained within a `try` block.
    - If an exception occurs within the `try` block, an `Exception` object is **created and thrown**.
    - Your code **can catch** this exception (using `catch`) and handle it in some rational manner.
- **System-generated exceptions** are automatically **thrown by the Java run-time system**.
- **Custom exceptions** are thrown 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.

Handling Exceptions (II)

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

- Every exception should be handled somewhere in your program.
- **If an exception has no handler, an error message is printed, and your program terminates**.
- You handle exceptions with the `try`/`catch` statement.
  - The `try` statement contains one or more statements that may cause an exception of the kind that you are willing to handle.
  - Each catch clause contains the **handler** for an exception type.

Handling Exceptions (III)

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage());
}
```

This constructor can throw a FileNotFoundException.

When an IOException is thrown, execution resumes here.

This is the exception that was thrown.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

A FileNotFoundException is a special case of an IOException.

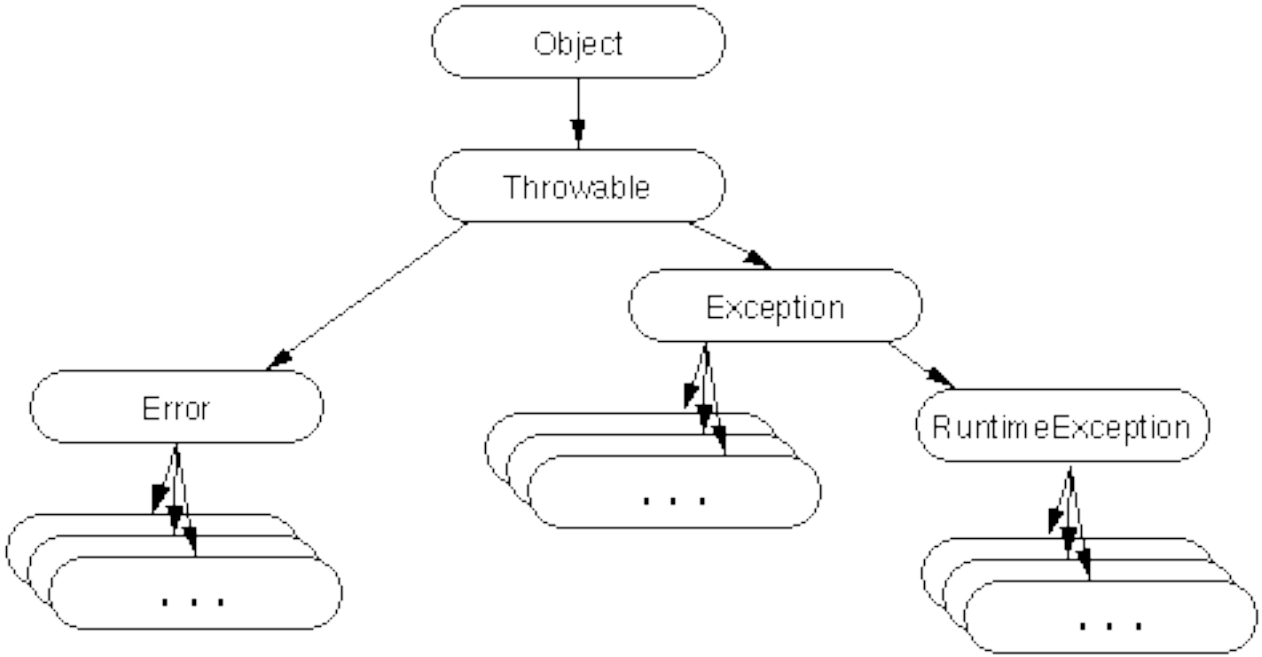
Exception Approach Advantage

```
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;
      } else
         errorCode = -3;
      close the file;
      if (theFileDidntClose && errorCode == 0)
         errorCode = -4;
      else
         errorCode = errorCode and -4;
   } else {
      errorCode = -5;
   }
   return errorCode;
}
```

```
readFile {
   try {
      open the file;
      determine its size;
      allocate that much memory;
      read the file into memory;
      close the file;
   } catch (fileOpenFailed) {
      doSomething;
   } catch (sizeDeterminationFailed) {
      doSomething;
   } catch (memoryAllocationFailed) {
      doSomething;
   } catch (readFailed) {
      doSomething;
   } catch (fileCloseFailed) {
      doSomething;
   }
}
```

`Throwable` Class (I)

- The `throw` statement requires a single argument: a throwable object.
  - Throwable objects are instances of any subclass of the `Throwable`

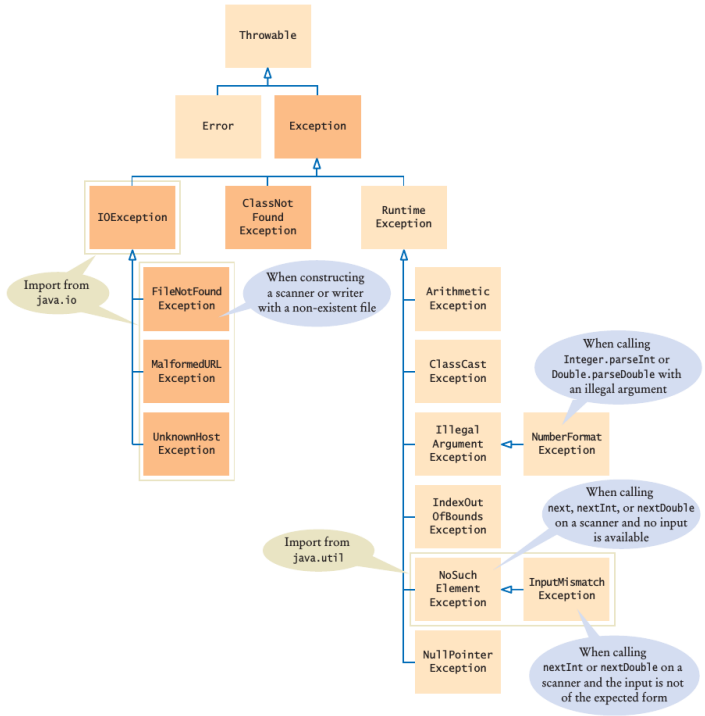


Exception Types (II)

- The **`Error` class represents more serious errors from which your program usually cannot recover**.
  - For example, there might be insufficient memory to execute a program.
  - Usually, you do not use or implement `Error` objects in your programs.
  - **A program cannot recover from `Error` conditions on its own**.
- The **`Exception` class comprises less serious errors** representing unusual conditions that arise while a program is running and from which **the program can recover**.
  - For example, `IllegalAccessException` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.
  - One `Exception` subclass has special meaning in the Java language: `RuntimeException`.

Exception Types (III)

- The **`RuntimeException` class represents exceptions that occur within the JVM** (during runtime).
- Examples
  - `NullPointerException` occurs when a method tries to access a member of an object through a null reference.
  - `InputMismatchException` class extends the `NoSuchElementException` class, which is used to indicate that the element being requested does not exist.



Exception Specification

- If a method throws an exception that it will not catch, but such exception will be caught by a different method.
  - You must use the keyword `throws` followed by an `Exception` type in the method header.

```
public class Main {
   private static final double[] price = {15.99, 27.88, 34.56, 45.89};
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.print("Enter item number: ");
      int item = input.nextInt();
      try{
         displayPrice(item);
      }
      catch(IndexOutOfBoundsException e){
         System.out.println("Price is $0");
      }
   }
   static void displayPrice(int item) throws IndexOutOfBoundsException {
      System.out.println("The price is $" + price[item]);
   }
}
```

Custom Exception (I)

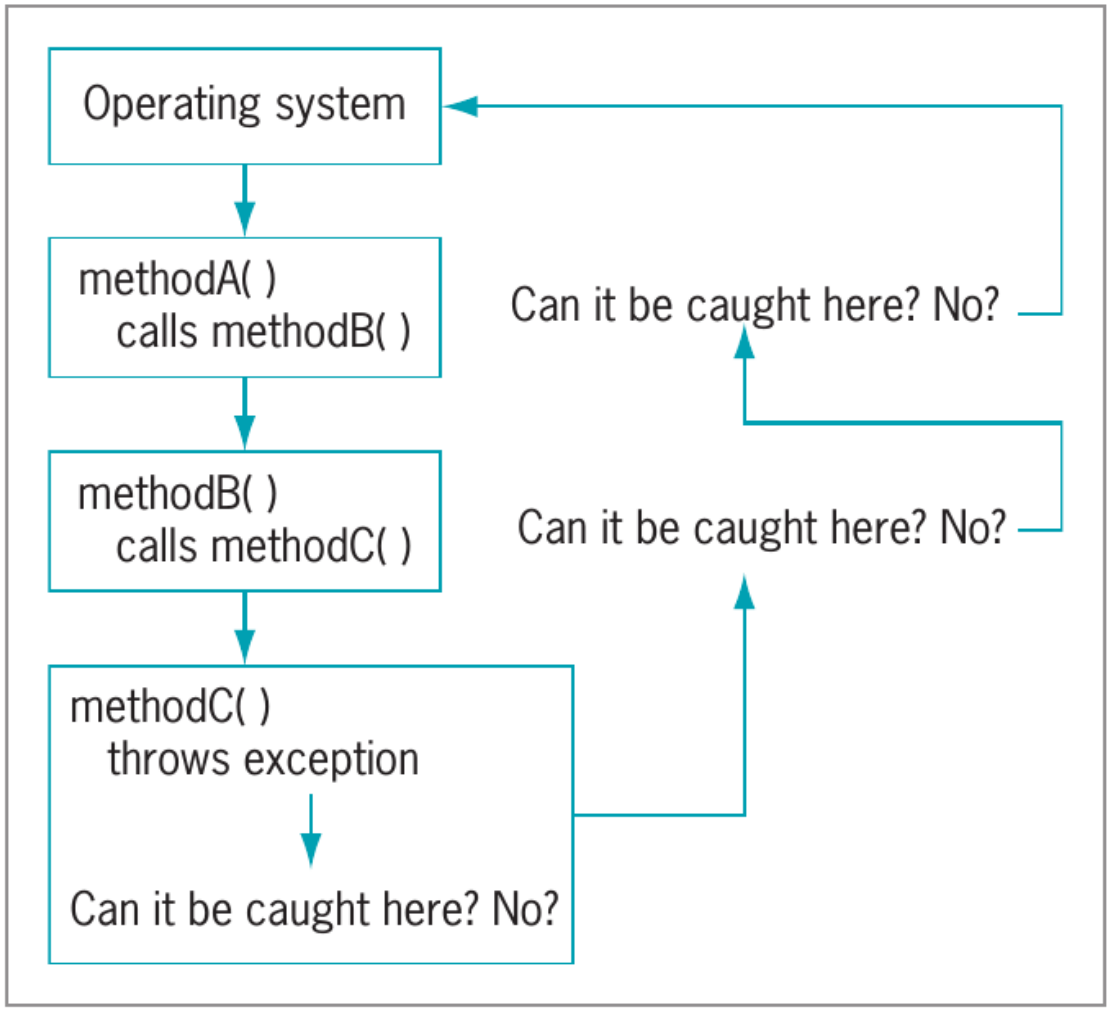
```
public class InvalidPriceException extends IllegalArgumentException{
   public InvalidPriceException(String s) {
      super(s);
   }
}
```

```
public class Product {
   private int id;
   private String description;
   private int price;
   ...
   public void setPrice(int price) {
      if(price > 0){
         this.price = price;
      }else{
         throw new InvalidPriceException(price+ " is not a valid price");
      }
   }
}
```

Custom Exception (II)

```
public static Product getProduct(){
      Scanner kbd = new Scanner(System.in);
      boolean flag = false;
      Product product = new Product();
      System.out.print("Id: ");
      int id = kbd.nextInt();
      kbd.nextLine();
      product.setId(id);
      System.out.print("Description: ");
      String description = kbd.nextLine();
      product.setDescription(description);
      int price = -1;
      do {
         try {
            flag = false;
            System.out.print("Price: ");
            int price = kbd.nextInt();
            kbd.nextLine();
            product.setPrice(price);
         }catch (InvalidPriceException e){
            flag = true;
         }
      }while(flag && price < 0);
      return product;
}
```

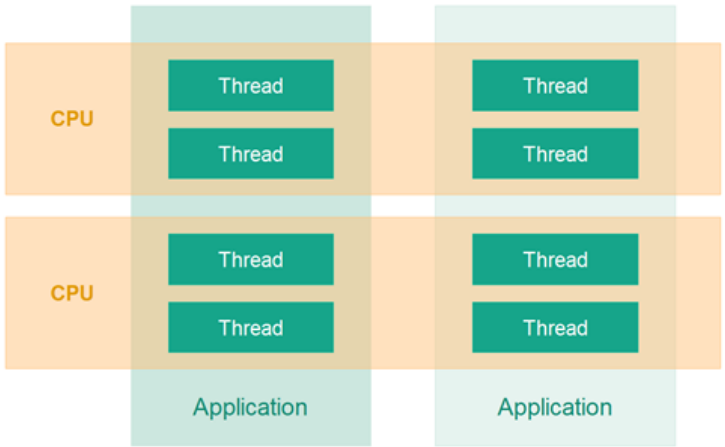
Tracing Exceptions



Java Threads

Multithreading

- Multithreading means that you **have multiple threads of execution inside the same application**.
- A thread is **like a virtual CPU executing your application**.
  - Thus, a multithreaded application is like an application that has multiple virtual CPUs executing different parts of the code at the same time.



- A single CPU will share its execution time among multiple threads, switching between executing each of the threads for a given amount of time.

Thread

- When a Java application is started its `main()` method is executed by the **main thread** - a special thread that is created by the Java VM to run your application.
- From inside your application you can create and start more threads which can execute parts of your application code in parallel with the main thread.
- Threads are instances of class `Thread`
- Java threads are objects like any other Java objects.

Creating and Starting Threads (I)

- Simple

```
Thread thread = new Thread(){
 public void run(){
   //executing code
 }
}
thread.start();
```

- Subclass

```
public class MyThread extends Thread {
 public void run(){
   //executing code
 }
}
```

```
MyThread myThread = new MyThread();
myTread.start();
```

Creating and Starting Threads (II)

- Simple

```
Runnable myRunnable = new Runnable(){
  public void run(){
    //executing code
  }
}
```

```
  Thread thread = new Thread(myRunnable);
  thread.start();
```

- Subclass

```
 public class MyRunnable implements Runnable {
  public void run(){
    //executing code
  }
}
```

```
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

Bibliography

Resources

- "Big Java: Early Objects", 6th Edition by Cay S. Horstmann
- "Java™:The Complete Reference", 7th Edition,Herbert Schildt
- "Java™Programming", 7th Edition, Joyce Farrell
- `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html`
- `http://beginnersbook.com/java-tutorial-for-beginners-with-examples/`
- `https://www.leepoint.net/index.html`
- `https://junit.org/junit5/docs/current/user-guide/`