Lecture 4

Exceptions and Classes

Course Website

- Slides + code from each lecture available on course website
- *Preliminary* course schedule (readings + exercises)
- Also has important announcements
- Check back often!

tiny.cc/tin213

Lab Pace

- Lab deadlines are on the course website
- You are **NOT** expected to complete one lab per lab session!
- More like one lab per month
- Don’t panic if everything you need to complete a lab hasn’t been covered before the first session for that lab!
- Next deadline: Sunday Dec. 15, 23:59

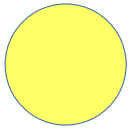
Bored of the Book Exercises?

- projecteuler.net
  - Math problems to solve with code
- codewars.com
  - All sorts of coding challenges
- adventofcode.com
  - One Christmas-themed code challenge per day until Christmas Eve, starting Dec. 1st
  - The Challenges of Christmas Past still available if you want to binge

Boolean Best Practices

Don't:

- `if(foo == true) …`
- `if(foo == false) …`
- `if(some expression) {`
  `  return true;`
  `} else {`
  `  return false;`
  `}`



Do:

- `if(foo) …`
- `if(!foo) …`
- `return some expression;`

Last Week's Bug

- Caused by a very common "problem": *mutability!*

```
public static void main(String[] args) {
    double[] p1 = {  0.0,  1.0, 0.0 };
    double[] p2 = {  1.0, -1.0, 0.0 };
    double[] p3 = { -1.0, -1.0, 0.0 };
    double   time = 0;

    Scene.createWindow(500, 500);
    while(true) {
        Scene.clearScene();
        p1 = transform(p1, time);
        p2 = transform(p2, time);
        p3 = transform(p3, time);
        Scene.drawTriangle(p1, p2, p3);
        Scene.present();
        Scene.wait(10);
        time += 0.01;
    }
}
```

Last Week's Bug

- Caused by a very common "problem": *mutability!*

We overwrite `p1`, etc. with the transformed point each loop step!

```
publ                                    args) {
                                        0.0 };
                                        0.0 };
                                        0.0 };
doubl          me = 0;

Scene          eateWindow(500, 500);
while          rue) {
    Scene.clearScene();
    p1 = transform(p1, time);
    p2 = transform(p2, time);
    p3 = transform(p3, time);
    drawTriangle(p1, p2, p3);
```

$$p1_1 = transform(p1_0, time_0)$$
$$p1_2 = transform(transform(p1_0, time_0), time_1)$$
$$p1_3 = transform(transform(transform(p1_0, time_0), time_1), time_2)$$
$$p1_4 = transform(transform(transform(transform(p1_0, time_0), time_1), time_2), time_3)$$
$$p1_5 = transform(transform(transform(transform(transform(p1_0, time_0), time_1), time_2), time_3), time_4)$$
$$p1_6 = transform(transform(transform(transform(transform(transform(p1_0, time_0), time_1), time_2), time_3), time_4), time_5)$$
$$\ldots$$

Avoid Mutation Whenever Possible

- What we intended: $p_n' = transform(p_n, time)$

```
public static void main(String[] args) {
    double[] p1 = {  0.0,  1.0, 0.0 };
    double[] p2 = {  1.0, -1.0, 0.0 };
    double[] p3 = { -1.0, -1.0, 0.0 };
    double   time = 0;

    Scene.createWindow(500, 500);
    while(true) {
        Scene.clearScene();
        double pt1 = transform(p1, time);
        double pt2 = transform(p2, time);
        double pt3 = transform(p3, time);
        Scene.drawTriangle(pt1, pt2, pt3);
        S          sent();
```

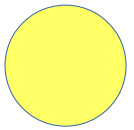
Don't *mutate* variables without a reason, prefer to *create new values* instead!

Immutable code is easier to read, write and debug!

More About Methods

```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y = a[1]*b[1];
    double z = a[2]*b[2];
    return x+y+z;
}
```

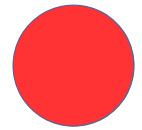
Method Head



```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y = a[1]*b[1];
    double z = a[2]*b[2];
    return x+y+z;
}
```

Comment: every method needs one. Describes method purpose, return value, argument, and *invariants*.

Method Head



```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y =  ]*b[1];
    double z
     eturn x
}
```

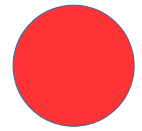
Method return type

Other classes can access this method

Method name

Argument list
(Comma-separated list of `type name`)

Method Body



```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y = a[1]*b[1];
    double z = a[2]*b[2];
    return x+y+z;
}
```

Return `x+y+z` to the caller

Arguments are *in scope* for the duration of the method

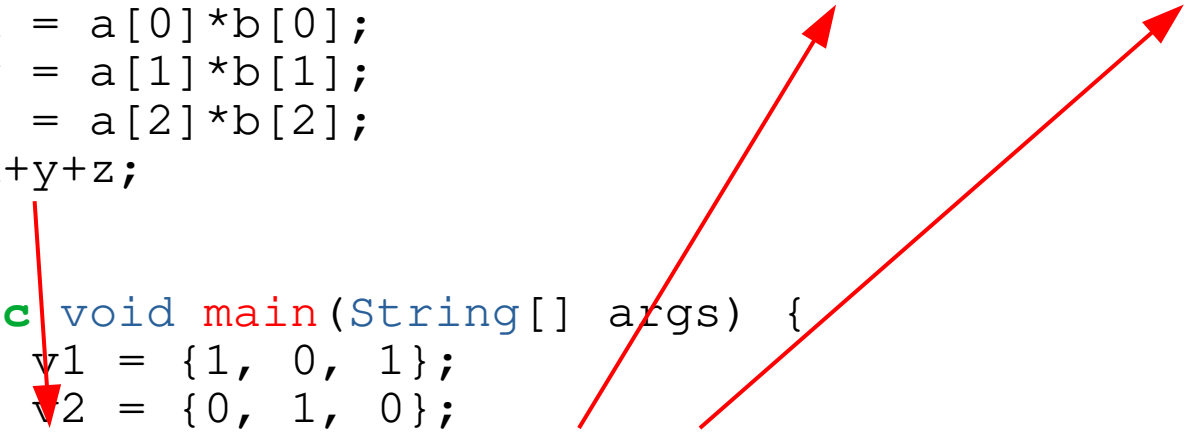
Methods are *Abstractions*

```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y = a[1]*b[1];
    double z = a[2]*b[2];
    return x+y+z;
}
```

Arguments may be *anything* (of the correct type) the caller decides to pass

Methods are *Abstractions*

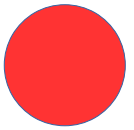
```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = a[0]*b[0];
    double y = a[1]*b[1];
    double z = a[2]*b[2];
    return x+y+z;
}

public static void main(String[] args) {
    double[] v1 = {1, 0, 1};
    double[] v2 = {0, 1, 0};
    double result = dotProduct(v1, v2)
}
```



- No connection between variable names in *caller* and *callee*.
- Caller variables are **not** in scope in callee, and vice versa.

Methods are *Abstractions*

```
/* Calculates the scalar product of two 3-dimensional vectors.
   a and b must be vectors in 3 dimensions, ordered {x, y, z}
 */
public static double dotProduct(double[] a, double[] b) {
    double x = v1[0]*v2[0];
    double y = v1[1]*v2[1];
    double z = v1[2]*v2[2];
    return x+y+z;
}

public static vo     ain(String[] args) {
    double[] v1          0, 1};
    double[] v           1, 0};
    double res           Product(v1, v2)
}
```



Compiler error: `v1` and `v2` are not in scope!

Array Recap

- ```
  String[] arr = new String[3];
  arr[0] = "an array";
  arr[1] = "which contains";
  arr[2] = "strings";
  ```

  or

- ```
  String[] arr = {
        "an array",
        "which contains",
        "strings"
  };
  ```

Array Recap

- Access elements by their *index*:

```
result[i] = arr1[i] + arr2[i];
```

- Arrays go well with `for`-loops:

```
for(int i = 0; i < arr.length; i++){
    doSomethingWith(arr[i]);
}
```

or

```
for(int elem: arr){
    doSomethingWith(elem);
}
```

Example: The Mean of a Million Numbers

New Tool: `Scanner`

- Can read `double`, `int`, etc. from `String`

```
import java.util.Scanner;

…

int age;
Scanner scan = new Scanner(System.in);
System.out.print("Enter your age: ");
age = scan.nextInt();
System.out.println("Wow, next year you'll be " + (age + 1));
```

```
valderman@maika:~$ java ScannerExample
Enter your age: 33
Wow, next year you'll be 34
valderman@maika:~$
```

- API ref: https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

Useful Terminal Tip

- `java Program < file.txt`
- Runs `Program` with `file.txt` as its input
  - `new Scanner(System.in)` will read from `file.txt` instead of keyboard input
  - Works on Mac/Linux/Windows
- Useful for “replaying” inputs
- Can also use `java Program > file2.txt` to save output to `file2.txt`
- Can be used together:
  `java Program < in.txt > out.txt`

Scanner *Consumes* its Input

```
Scanner scan = new Scanner(System.in);
System.out.println(scan.nextInt());
System.out.println(scan.nextInt());
System.out.println(scan.nextInt());
System.out.println("Bye!");
```

Contents of `input.txt`:

```
5
10
42
110
...
```

Output of `java ScannerTest < input.txt`:

```
5
10
42
Bye!
```

Exceptions

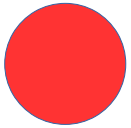
An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
(Definition from https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html)

We can **catch** exceptions in order to handle them gracefully.

If an exception happens and is not caught, then the program crashes.

Catching Exceptions

```
try {
  block 0
} catch (ExceptionType1 varName1) {
  block 1
} catch (ExceptionType2 varName2) {
  block 2
} … catch (ExceptionTypen varNamen) {
  block n
} finally {
  block n+1
}
```



1) Block 0 is executed.
2) If no exception occurs: block n+1 is executed.
3) If an exception of type `ExceptionType` occurs while executing block 1, then the exception is stored in the variable (“caught”), and block 2 is executed, then block n+1 is executed.
4) If an exception of another type occurs, block n+1 is executed, then the exception is thrown to the calling method. (If this is in `main`: the program crashes.)

The part **`finally`** `{` block n+1 `}` is optional

Example: A Safer Phone Bill Calculator

Catch or Specify

We can **specify** which exceptions a method might throw:

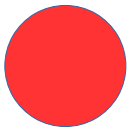
```
public static double openFile() throws FileNotFoundException {
```

Now, if a method calls `openFile()`, then it must catch the `FileNotFoundException` or specify it too. Otherwise, the compiler will report an error.

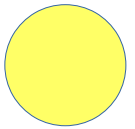
Catch or Specify

Three types of exception:

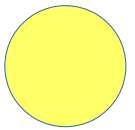
- *Checked exceptions*
  Conditions that a well-written program should recover from.
  Example: `FileNotFoundException`
  Must be either handled using try...catch or specified



- *Errors*
  Subclasses of `Error`
  Exceptions caused by something external to the program
  Example: `IOError`
  Choose whether to handle or let program crash



- Runtime exceptions
  Subclasses of `RuntimeException`
  Usually indicate a bug
  Example: `NullPointerException`
  Program should usually crash



Bad Practices

Don’t use a single catch-all `catch` clause!
(“Pokémon exception handling”)

```
try {
   …
} catch (Exception e) {
   …
}
```

Instead:

- One `catch` per exception…
- ...and *only* for the exceptions you know how to deal with!

Bad Practices

```
try {
    …
} catch (FileNotFoundException e) {
  throw e;
}
```

Don’t just throw an exception again (Instead: remove the catch block and specify `FileNotFoundException`)

```
try {
   …
} catch (FileNotFoundException e) {
}
```

Don’t ignore an exception (**swallowing** an exception)

Two Schools of Thought

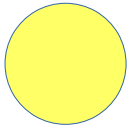
- Look Before You Leap (LBYL)

```
if (! scan.hasNextDouble()) {
   x = scan.nextDouble();
} else {
   System.out.println("Please enter a number");
}
```

- Easier to Ask Forgiveness than Permission (EAFP)

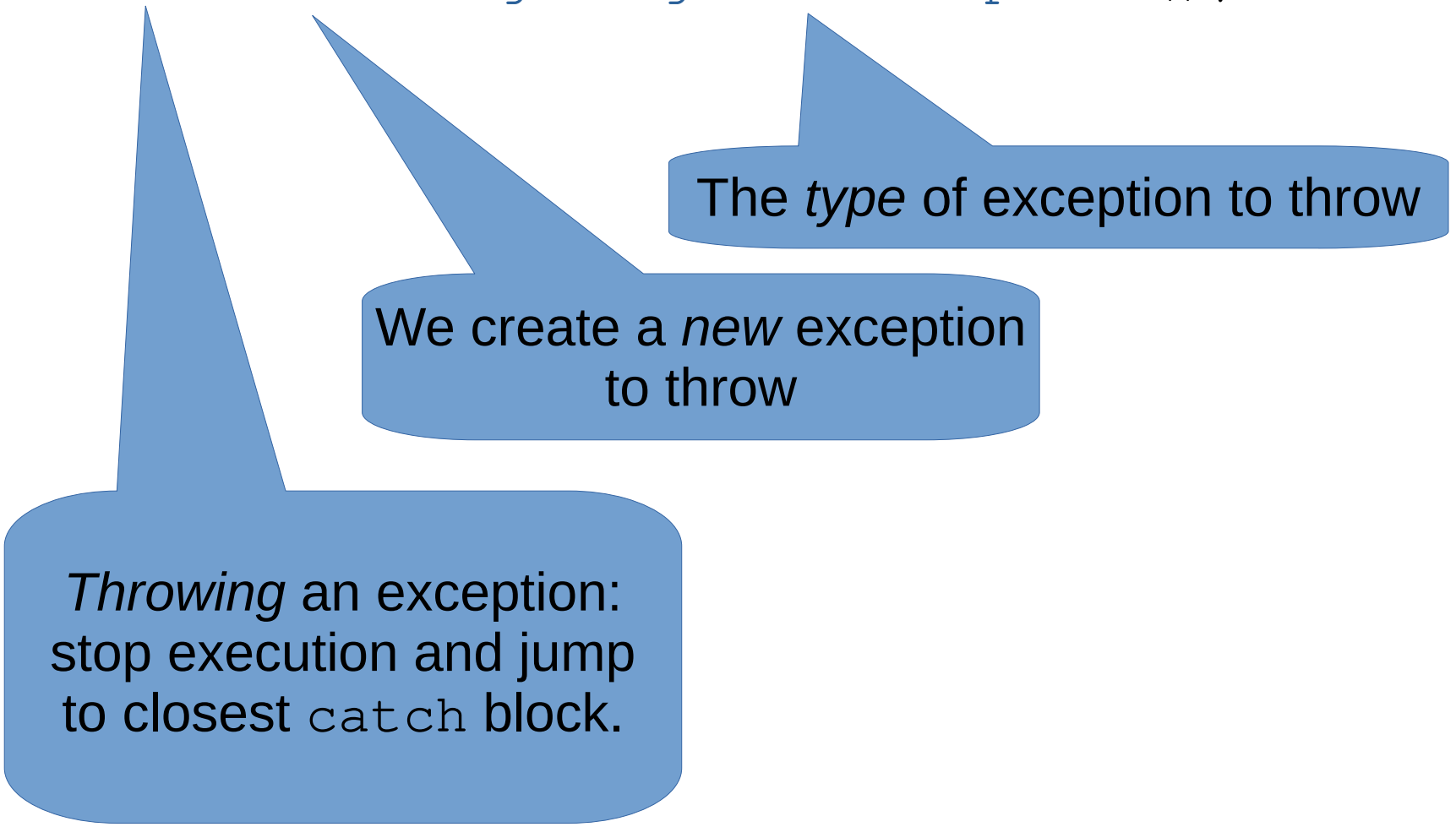
```
try {
   x = scan.nextDouble();
} catch (NumberFormatException e) {
   System.out.println("Please enter a number");
}
```

Don’t overuse exceptions, use LBYL where possible



Throwing Exceptions

```
throw new IllegalArgumentException();
```



What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

What's the Output?

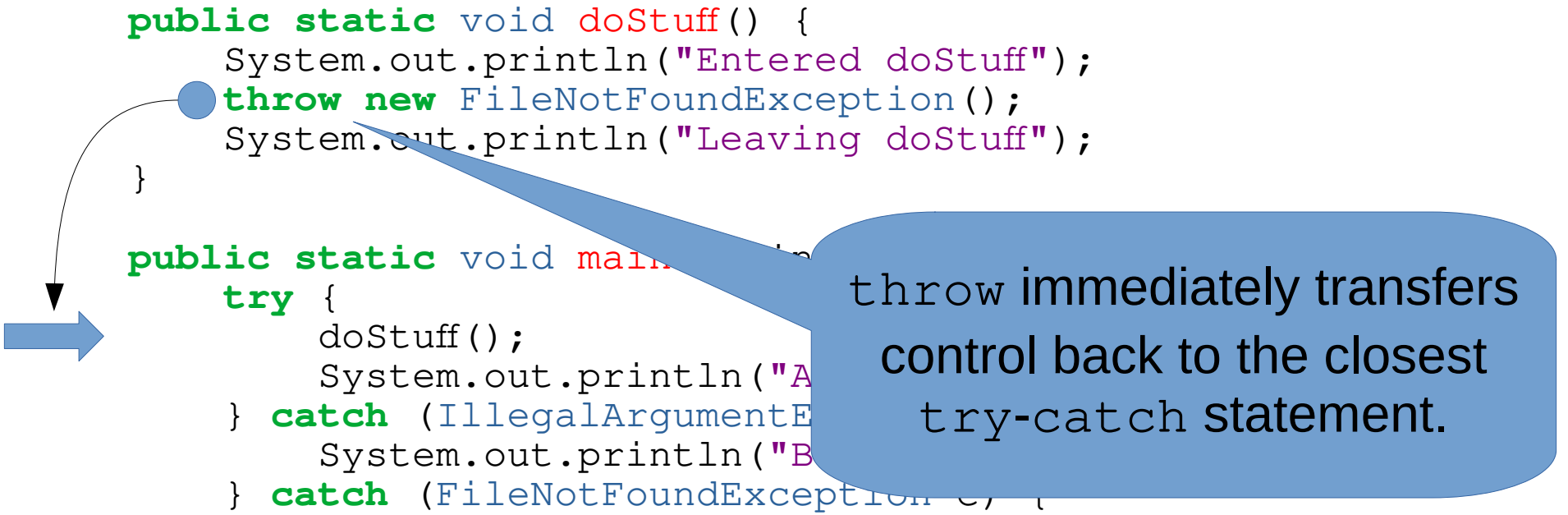
```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main
    try {
        doStuff();
        System.out.println("A
    } catch (IllegalArgumentE
        System.out.println("B
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```



What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
 ???} catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

Exception types are checked in order

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] ar
    try {
        doStuff();
        System.out.println("After do uff");
    } catch (IllegalArgumentExcepti n e) {
        System.out.println("Bad argument!");
!!!} catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

The first matching block is executed

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

What's the Output?

```
public static void doStuff() {
     System.out.println("Entered doStuff");
     throw new FileNotFoundException();
     System.out.println("Leaving doStuff");
}

public static void ma
     try {
          doStuff();
          System.out.pr
     } catch (IllegalAr
          System.out.println                gument!");
     } catch (FileNotFou          ption e) {
          System.out.pr        n("Oh no, the file was not found!");
     } catch (Arith      icException e) {
          System.out.println("Division by zero, most likely!");
     } finally {
          System.out.println("Finally!");
     }
     System.out.println("Bye bye!");
}
```

finally block is *always* executed, either after a catch block or after the try block.

What's the Output?

```
public static void doStuff() {
    System.out.println("Entered doStuff");
    throw new FileNotFoundException();
    System.out.println("Leaving doStuff");
}

public static void main(String[] args) {
    try {
        doStuff();
        System.out.println("After doStuff");
    } catch (IllegalArgumentException e) {
        System.out.println("Bad argument!");
    } catch (FileNotFoundException e) {
        System.out.println("Oh no, the file was not found!");
    } catch (ArithmeticException e) {
        System.out.println("Division by zero, most likely!");
    } finally {
        System.out.println("Finally!");
    }
    System.out.println("Bye bye!");
}
```

Throwing Exceptions

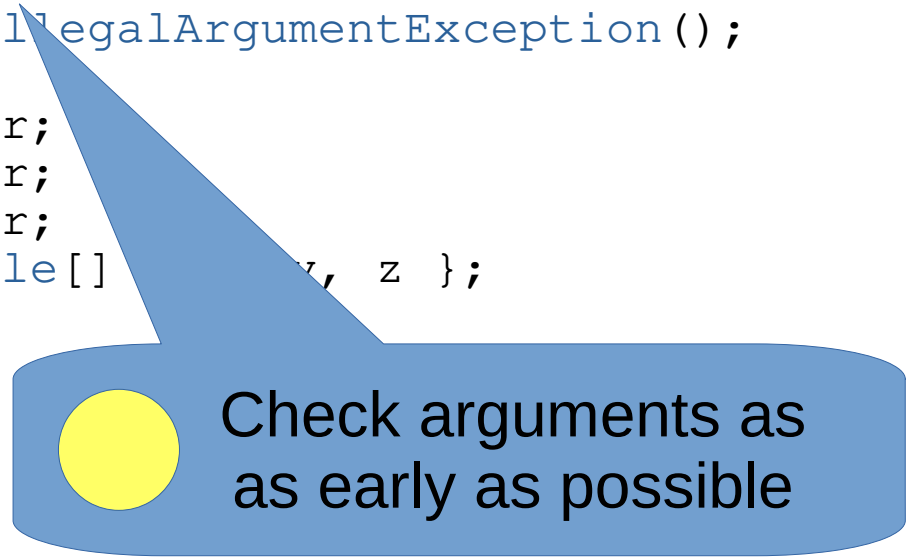
- Common use case: checking method arguments

```
public static double divide(double[] a, double divisor) {
    double x, y, z;
    if(Math.abs(divisor) < 0.0001) {
        throw new IllegalArgumentException();
    }
    x = a[0]/divisor;
    y = a[1]/divisor;
    z = a[2]/divisor;
    return new double[] { x, y, z };
}
```

Throwing Exceptions

- Common use case: checking method arguments

```
public static double divide(double[] a, double divisor) {
    double x, y, z;
    if(Math.abs(divisor) < 0.0001) {
        throw new IllegalArgumentException();
    }
    x = a[0]/divisor;
    y = a[1]/divisor;
    z = a[2]/divisor;
    return new double[]      y, z };
}
```



Throwing Exceptions

- Common use case: checking method arguments

```
public static double divide(double[] a, double divisor)
        throws IllegalArgumentException {
    double x, y, z;
    if(Math.abs(divisor) < 0.0001) {
        throw new IllegalArgumentException();
    }
    x = a[0]/divisor;
    y = a[1]/divisor;
    z = a[2]/divisor;
    return new double[] { x, y, z
}
```

If it's a checked exception, don't forget to specify it!

`IllegalArgumentException` is *not* checked, but it's a good idea to specify it anyway.

Never Trust User Input!

- Write code as if the user is either stupid, malicious, or both!
- If something can go wrong, it *will* go wrong.
- If something can't possibly go wrong, it will go wrong anyway...

A typical user

- ...and usually in the worst way possible!
- *Always* validate user input!

`null`

- Null is a “special” value denoting nothing.
- *Any* string, array or objects can be `null`
- `int`, `double`, `char`, etc. can’t
- Often used to signal “no input” or similar

`null`

- Called “my billion dollar mistake” by its inventor
- Has been around since 1965(!)
- Now considered a “code smell”
- Avoid using it whenever possible
- We’ll see more safe/modern alternatives later

Tony Hoare,
Inventor of `null`

Classes

Java is an *object-oriented* language. The focus is on:

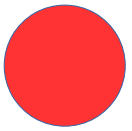
- **objects** not actions
- **data** not logic

You can *design your own types* by writing **classes**. This is intended to be your main tool for solving problems.

Classes

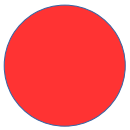
- A **class** has **fields**, **methods**, and **constructors**.
- When we have a class `C`, there can be many **objects** or **instances** of type `C` - a class is a *blueprint* for objects!
- The **fields** say what data each object holds.
- The **methods** say what can be done with an object of type `C`.
- The **constructors** say how we can create an object of type `C`.

Classes - Syntax



```
public class ClassName {
  private type1 field1;
  private type2 field2;
  …
  private typen fieldn;

  public ClassName(type1 field1, …, typen fieldn) {
    this.field1 = field1;
    …
    this.fieldn = fieldn;
  }

  public type method1(type arg, …, type arg) {
    …
  }

  public type method2(type arg, …, type arg) {
    …
  }
}
```

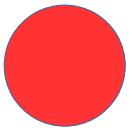
Classes - Syntax



```
public class ClassName {
  private type1 field1;
  private type2 field2;
  …
  private typen fieldn;

  public ClassName(type1 …, typen fieldn) {
    this.field1 = field1;
    …
    this.fieldn = fieldn;
  }

  public type method1(type arg, …, type arg) {
    …
  }

  public type method2(type arg, …, type arg) {
    …
  }
}
```

Fields: hold data

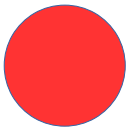
Classes - Syntax



```
public class ClassName {
  private type1 field1;
  private type2 field2;
  …
  private typen fieldn;

  public ClassName(type1 field1, …, typen fieldn) {
    this.field1 = field1;
    …
    this.fieldn = fieldn;
  }

  public type method1(type arg, …, type arg) {
    …
  }

  public type method2                    type arg) {
    …
  }
}
```

Methods: operations on fields and arguments

Classes - Syntax

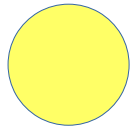


```
public class ClassName {
  private type1 field1;
  private type2 field2;
  …
  private typen fieldn;

  public ClassName(type1 field1, …, typen fieldn) {
    this.field1 = field1;
    …
    this.fieldn = fieldn;
  }

  public type method2(type arg, …, type arg) {
    …
  }
}
```

Constructors: special methods describing how to create new instances of the class

Access Modifiers

- If a method of class C is `public`, then other classes can call it.
- If a method of class C is `private`, then only class C can call it.
- Make methods `private` unless you have a good reason to make them `public`.



Getters and Setters

Given a field:

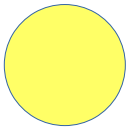
- a **getter** is a method that returns the value of the field

```
public double getRadius() {
   return radius;
}
```

- a **setter** is a method that changes the value of the field

```
public void setRadius(double r) {
   radius = r;
}
```

- Fields should always be `private`. Other classes should use getters and setters.



Getters and Setters

Given a field:

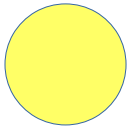
- a **getter** is a method that returns the value of the field

```
public double getRadius() {
  return radius;
}
```

- a **setter** is a method that changes the value of the field

```
public void setRadius(double r) {
  if(r < 0) {
    throw new IllegalArgumentException("negative radius");
  }
  radius = r;
}
```

- Setters should enforce any *invariants* on the field.
- Design goal: make it impossible to represent invalid values.



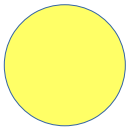
Class Methods

The word static means a method is a *class method*. It belongs to the class, not the instances of the class.

We call it by `ClassName.method()`, not `object.method()`

A class method cannot access instance fields.

Object-oriented Philosophy

- Small is beautiful 
  - Lots of small classes instead of a few big ones
  - In a class, lots of short methods, not a few long ones

Reading and Exercises

- Reading
  - Java Direkt med Swing Chapter 2, plus Sections 3.8, 11.4
  - Code Complete Chapter 6
- Exercises
  - Get started on laboratory 2
  - Java Direkt med Swing Exercises 2.1, 2.2, 2.3