Introduction to Programming

05 Object Orientation II

Stephan Krusche

29 November 2023
Technical University of Munich

Schedule

#	Date	Subject
1	18.10.23	Introduction
1b	25.10.23	Central exercise
	01.11.23	**No lecture**
2	08.11.23	Control Structures
3	15.11.23	Data Types
4	22.11.23	Object Orientation I
5	**29.11.23**	**Object Orientation II**
6	06.12.23	Object Orientation III
7	13.12.23	Algorithms
	20.12.23	**No lecture**
8	10.01.24	Programming Languages
9	17.01.24	Graphical User Interfaces
10	24.01.24	Recursion
11	31.01.24	Beyond Programming
12	07.02.24	Course Review

Roadmap of today's lecture

- **Context**
  - Apply the basics of object oriented programming
  - Use basic control structures (**`if`**, **`switch`**, **`for`**, **`while`**)
  - Implement and use basic data types (**`List`**, **`Stack`**, **`Queue`**)
  - Apply abstraction, encapsulation, inheritance and polymorphism
- **Learning goals**
  - Understand the idea behind generic types and polymorphic methods
  - Use generics with predefined data types
  - Create generic types
  - Wrap primitive data types in objects
  - Use predefined collection data types such as List, Map and Set with for each loops
  - Throw and catch exceptions

Outline

- **Generics (part 1)**
- Generics (part 2)
- Object data types
- Error handling

Example

Subclass of **`Object`**

```
class Poly {
    public String toString() {
        return "Hello";
    }
}

public class PolyTest {

    public static String addWorld(Object obj) {
        return obj.toString() + " World!";
    }

    public static void main(String[] args) {
        Object poly = new Poly();
        System.out.println(addWorld(poly));
    }
}
```

Implicitly contains the empty constructor

Class method which can be invoked with any **`Object`**

Dynamic type (runtime type)

A variable of class **`A`** can be assigned an object of **any subclass** of **`A`**

Static type (compile time type)

```
Output: Hello world!
```

Example

```java
class Poly {
    public String greeting() {
        return "Hello";
    }
}

public class PolyTest {

    public static void main(String[] args) {
        Object poly = new Poly();
        System.out.println(poly.greeting() + " World!");
    }
}
```

Dynamic type (runtime type)

Static type (compile time type)

The variable **`poly`** is declared as **`Object`** (**static** type)

The compiler does not know whether the current type (**dynamic** type) is a subclass in which **`greeting()`** is defined

```
→ Compile error
error: cannot find symbol
        System.out.println(poly.greeting() + " World!");
                               ^
  symbol:   method greeting()
  location: variable poly of type Object
1 error
```

Workaround

- Use an explicit **cast** to the appropriate subclass

```
class Poly {

    public String greeting() {
        return "Hello";
    }
}

public class PolyTest {

    public void main(String[] args) {
        Object poly = new Poly();
        if (poly instanceof Poly) {
            System.out.print(((Poly) poly).greeting() + " World!\n");
        }
        else {
            System.out.print("Sorry: no cast possible!\n");
        }
    }
}
```

Check if **`poly`** is an instance of **`Poly`**, i.e. has the dynamic type **`Poly`** (or one of **`Poly`**'s subclasses)

Cast **`poly`** to the static type **`Poly`**

Static type vs. dynamic type

- A variable `x` of a class `A` can take objects `b` from all subclasses `B` of `A`
- By this assignment, Java forgets the membership of `B`, because Java treats all values of x as objects of the class `A` (**static** type)
- We can use the expression `x instanceof B` to test the class membership of `x` at runtime (**dynamic** type)
- If we are sure that `x` is from class `B`, we can **cast** to that type
- If the current value of the variable `x` is indeed an object (of a subclass) of class `B` when checked, the expression returns exactly that object
- Otherwise, an **exception** is thrown

Generics

- Format: generics use **< >** to specify parameter types in generic class creation

Diamond operator

```
// To create an instance of generic class
List<Type> list = new ArrayList<Type>()
```

```
// Example
List<String> list = new ArrayList<String>()
```

Generics: parameterized types

- Create classes that **work with different data types**
- Allow types (**`Integer`**, **`String`**, etc, and user defined types) to be used as parameter to methods, classes, and interfaces
- Classes, interfaces, or methods that operate on parameterized types are called **generic** entities
- **`Object`** is the superclass of all other classes and an **`Object`** reference can refer to any type of object
- Many built-in classes in Java use generics (e.g. **`List<E>`**, **`Set<E>`**, **`Map<K, V>`**)

Example

```
class Course<T> {
    T obj;

    Course(T obj) {
        this.obj = obj;
    }
    public T getObject()  {
        return this.obj;
    }

    public static void main(String[] args) {
        Course <String> courseObj = new Course<>("INFUN");
        System.out.println("Course: " + courseObj.getObject());

        Course <Integer> periodObj = new Course<>(2023);
        System.out.println("Semester: " + periodObj.getObject());
    }
}
```

We use **< >** to specify Parameter type, **T** is used by convention, but it is possible to use any character

An object of type **T** is declared

instance of **String** type

instance of **Integer** type

```
Course: INFUN
Semester: 2023
```

Generics only work with reference types

- When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type
- We cannot use primitive data types like **`int, char`**

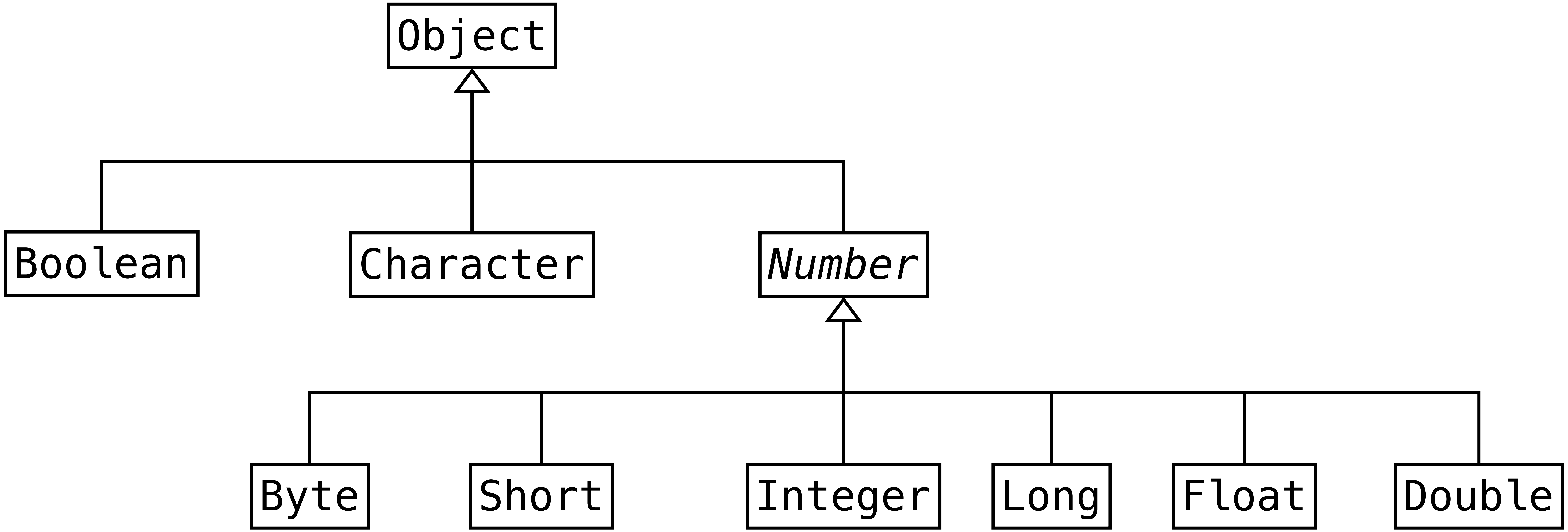
```
Course <int> intCourse = new Course<int>(2022);
System.out.println(intCourse.getObject());
```

Type argument **cannot** be of primitive type

- The above line results in a **compile time error**, that can be resolved by using type wrappers to encapsulate a primitive type

```
java: unexpected type
  required: reference
  found:    int
```

Wrapper classes



Generics only work with reference types

- Remember, generics are a **compile time feature**, meaning the type parameter is erased and all generic types are implemented as type **`Object`**
- Arrays can be passed to the type parameter because they are reference types

```
Course <int[]> intCourse = new Course<>(new int[] { 2021, 2022 });
System.out.println(Arrays.toString(intCourse.getObject()));
```

L05E02 Generic Points

Not started yet.

Start exercise

Easy

Due date tonight

10 min

3 pts

- Problem statement
  - Create a class **`Point`** using a generic data type for the **`x`** and **`y`** value
  - This allows users of the **`Point`** class to instantiate objects with **`Integer`**, **`Float`**, **`Double`**, …
  - Create a constructor with two parameters **`x`** and **`y`**
  - Create several point objects

Example solution

```
class Point<T> {
    T x;
    T y;

    Point(T x, T y) {
        this.x = x;
        this.y = y;
    }
}
```

Usage

```
public static void main(String[] args) {
    Point<Integer> integerPoint = new Point<>(1, 2);
    Point<Double> doublePoint = new Point<>(1.5, 2.3);
}
```

Example solution (with comments)

```java
class Point<T> {
    T x;
    T y;

    Point(T x, T y) {
        this.x = x;
        this.y = y;
    }
}
```

Two attributes with the generic type

Usage

```java
public static void main(String[] args) {
    Point<Integer> integerPoint = new Point<>(1, 2);
    Point<Double> doublePoint = new Point<>(1.5, 2.3);
}
```

Break

10 min

The lecture will continue at **15:00**

Outline

- Generics (part 1)
- **Generics (part 2)**
- Object data types
- Error handling

Type erasure

- Generics were introduced to the Java language to provide **tighter type checks** at **compile time** and to support generic programming
- To implement generics, the Java compiler applies **type erasure** to
  - Replace all type parameters in generic types with their bounds or **`Object`** if the type parameters are unbounded
  - Insert **type casts** if necessary to preserve type safety
  - Generate **bridge methods** to preserve **polymorphism** in extended generic types
- **Type erasure** ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead

- https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

Type erasure

```java
public <T> List<T> genericMethod(List<T> list) {
    // ...
}
```

```java
public List<Object> genericMethod(List<Object> list) {
    // ...
}
```

For illustration

```java
public List genericMethod(List list) {
    // ...
}
```

which in practice results in

Multiple types

- We can also write generic functions that can be called with different types of arguments
- Based on the type of arguments passed, the compiler handles each method

3 generic types

A generic method

```
class Tutors {
    static <T, I, D> void genericDisplay(T name, I id, D domain) {
        System.out.println("email: " + name + id + domain);
    }

    public static void main(String[] args) {
        genericDisplay("lucas", 1234, "@tum.de");
        genericDisplay("felix", 4321, "@tum.de");
        genericDisplay("tim", 7890, "@tum.de");
    }
}
```

It is possible to pass multiple type parameters

Calling generic method with arguments **`String, Integer, String`**

```
email: lucas1234@tum.de
email: felix4321@tum.de
email: tim7890@tum.de
```

Benefits

+ **Code reuse**: write a method / class / interface once and use it for any type
+ Enable implementation of **generic classes and algorithms**
+ **Stronger checks** at **compile time**
+ Individual **type casting is not needed**
+ **Type safety**: generics make errors to appear during compile time instead of at runtime

Type safety

- Generics make errors appear during compile time instead of at runtime

```java
import java.util.*;

class Test {
    public static void main(String[] args) {
        List list = new ArrayList();

        list.add("Sachin");
        list.add("Rahul");
        list.add(10);

        String s1 = (String)list.get(0);
        String s2 = (String)list.get(1);
        String s3 = (String)list.get(2);
    }
}
```

Compiler allows this

```java
import java.util.*;

class Test {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        list.add("Sachin");
        list.add("Rahul");
        list.add(10);

        String s1 = list.get(0);
        String s2 = list.get(1);
        String s3 = list.get(2);
    }
}
```

The compiler doesn't allow this

Output

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String at
Test.main(Test.java:13)
```

Bounded generics

- For a type parameter T, you can also specify a superclass or an interface that T should implement in any case

```
public class ExecutableList<E extends Executable> {
    E element;
    ExecutableList<E> next;

    void executeAll() {
        element.execute();
        if (next != null) {
            next.executeAll();
        }
    }
}
```

Bounded generics

- Note that **`extends`** is also used for bounded generics
- Further restrictions apply here, such as a parameterized class can be a superclass
- Interfaces can also be parameterized
- In particular, **`Comparable<T>`** can be parameterized - with the class whose objects you want to compare with
- Example

```
public class Test implements Comparable<Test> {
    public int compareTo (Test x) {
        return 0;
    }
}
```

Bounded generics

- Bounded means restricted: restrict the types that a generic accepts
- For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type and all its superclasses (lower bound)
- To declare an upper-bounded type, we use the keyword extends after the type, followed by the upper bound that we want to use

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
  //...
}
```

Any **Number** type

L05E03 Generic Animal Shelter

Not started yet.

Start exercise

Medium

Due date tonight

15 min

4 pts

- Problem statement: create a generic **`Shelter`** for animals
  - Implement a new class **`Shelter`** with a bound generic that only accepts animal sub classes
  - The shelter has a limited capacity
  - Implement a method **`addAnimal(T animal)`** to add animals to the shelter if the limit is not yet reached
  - Implement a method **`makeAllAnimalsSound()`** which invokes **`makeSound()`** on each animal

```java
abstract class Animal {
    private final String name;
    /* ... */
    public abstract String makeSound();
}

final class Dog extends Animal { /* ... */ }

final class Cat extends Animal { /* ... */ }
```

Example solution

```java
class Shelter<T extends Animal> {
    private final List<T> animals = new ArrayList<>();
    private final int capacity;

    public Shelter(int capacity) {
        this.capacity = capacity;
    }

    public void addAnimal(T animal) {
        if (animals.size() < capacity) {
            animals.add(animal);
            System.out.println(animal.getName() + " has been added to the shelter.");
        } else {
            System.out.println("Shelter is full. Cannot add " + animal.getName());
        }
    }

    public void makeAllAnimalsSound() {
        for (T animal : animals) {
            System.out.println(animal + " says " + animal.makeSound());
        }
    }
}
```

Example solution (with comments)

```java
class Shelter<T extends Animal> {
    private final List<T> animals = new ArrayList<>();
    private final int capacity;

    public Shelter(int capacity) {
        this.capacity = capacity;
    }

    public void addAnimal(T animal) {
        if (animals.size() < capacity) {
            animals.add(animal);
            System.out.println(animal.getName() + " has been added to the shelter.");
        } else {
            System.out.println("Shelter is full. Cannot add " + animal.getName());
        }
    }

    public void makeAllAnimalsSound() {
        for (T animal : animals) {
            System.out.println(animal + " says " + animal.makeSound());
        }
    }
}
```

Generic bounded type

Add subclasses of **`Animal`**

Use specific features of **`Animal`**

Use specific features of **`Animal`**

30 min

The lecture will continue at **16:10**

Outline

- Generics (part 1)
- Generics (part 2)
- **Object data types**
- Error handling

Wrapper classes

- Besides the constructor **`public Integer(int value)`** there is also **`public Integer(String s) throws NumberFormatException`**
  - This constructor returns an **`Integer`** object for a **`String`** object **`s`**
- **`public boolean equals(Object obj);`**
  returns **`true`** exactly if **`obj`** contains the same **`int`** value
- Similar wrapper classes exist for the other base types

Features

- All wrapper classes for types **type** (except **char**) have
  - Constructors from base values or string objects
  - A static method **`parseType(String s)`**
  - A method boolean **`equals(Object obj)`**
- Except for **boolean**, all have constants **MIN_VALUE** and **MAX_VALUE**
- **Character** contains further auxiliary functions, e.g., to recognize digits, to convert lowercase letters into uppercase letters
- Numeric wrapper classes are combined in the common superclass **Number**
  - This class is **abstract** i.e. you cannot create any **Number** objects

Specialties

- **`Double`** and **`Float`** additionally contain the constants

```
NEGATIVE_INFINITY = –1.0/0
POSITIVE_INFINITY = +1.0/0
NaN               = 0.0/0
```

- Additionally, there are the tests for infinity of values
  - `public static boolean isInfinite(double v);`
  - `public static boolean isNaN(double v);`
    (analog for float)
  - `public boolean isInfinite();`
  - `public boolean isNaN();`

`String` (object methods)

- **`int indexOf(String string)`**
  find the index of the first occurrence of a character or a string
- **`int indexOf(String string, int fromIndex)`**
  find the index of the first occurrence of a character or a string starting at **`fromIndex`**
- **`char charAt(int index)`**
  get the character at the specified **`index`**
- **`String replace(char oldChar, char newChar)`**
  replace all occurrences of **`oldChar`** with **`newChar`**
- **`String substring(int beginIndex, int endIndex)`**
  get the part of the string from **`beginIndex`** to **`endIndex`**
- **`String[] split(String regex, int limit)`**
  split the string based on the given **`regex`** (regular expression)
- **`String strip()`**
  remove all trailing and leading whitespaces

https://www.crio.do/blog/string-methods-in-java

Splitting a string

```
String csv = "1,2,3,4";
String[] values = csv.split(",");
int sum = 0;
for (String value : values) {
    sum += Integer.parseInt(value);
}
System.out.println(sum);
```

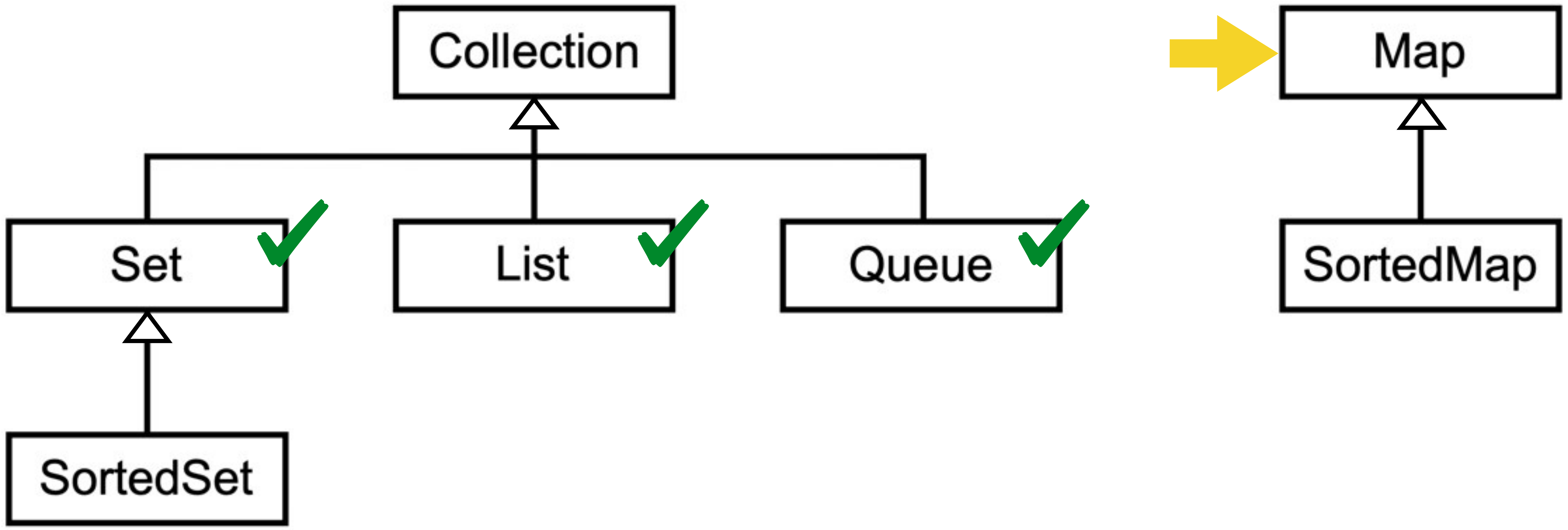
for each loop: easier and more safe than traditional for loops, so they should be preferred

Output

```
10
```

Java collections framework

- Unified architecture for representation and manipulation of collections
- Abstraction from the implementation of collections
- Reduction of programming effort
- Increase in the performance of operations on collections
- Interoperability of independent collections



http://download.oracle.com/javase/tutorial/collections/index.html

java.util.Map<K, V>

Generic type

- The **Map** interface maps keys of type K to values of type V (models a mathematical function)
- Contains each key only once and maps the key to only one value
- Offers different implementations
- Example: **HashMap**

```
import java.util.*;

class MyPlayground {
    public static void main(String[] args) {
        Map<String, Integer> wordCount = new HashMap<>();
        wordCount.put("University", 10);
        wordCount.put("Heilbronn", 5);

        System.out.println(wordCount.size());
        System.out.println(wordCount.get("University"));
        System.out.println(wordCount.get("München"));

        for (String word : wordCount.keySet()) {
            Integer count = wordCount.get(word);
            System.out.println(word + ": " + count);
        }
    }
}
```

It's best practice to use the **interface** for the **static** type and the **implementation** for the **dynamic** type: this allows to easily exchange the implementation when needed (e.g. with a **LinkedHashMap**)

for each loop

Output

```
2
10
null
University: 10
Heilbronn: 5
```

`java.util.Map<K,V>`

- **`int size()`**: return the length of the map
- **`boolean isEmpty()`**: checks if the map is empty
- **`V put(K key, V value)`**: adds the given value for the given key (potentially overriding existing values)
- **`V get(Object key)`**: gets the value for the given key
- **`V remove(Object key)`**: removes the given key (and its value)
- **`boolean containsKey(Object key)`**: checks if the map contains the given key (with a value)
- **`boolean containsValue(Object value)`**: checks if the map contains the given value (with a key)
- **Set<K> keySet()**: returns all keys in a Set (useful for iterating with for each loops)
- **`Collection<V> values()`**: returns all values in a collection (useful for iterating with for each loops)
- **`void clear()`**: removes all keys with their values

Exercise L05E04

10 min

- Copy the following **static method** that calculates the **factorial** number of a positive integer iteratively (using a **`for` loop**)

```
public class Playground {
   private static long factorial(long number) {
        long fact = 1;
        for(int i = 1; i <= number; i++) {
           fact = fact * i;
        }
        return fact;
    }
}
```

- To improve the performance of the calculation, implement a caching mechanism based on **`HashMap<Long, Long>`**
- Store the **input** as a **key** and the **output** as a **value** in the map
- In case the **cache** (map) includes the calculation, you can return immediately
- Otherwise, calculate the result, store it in the **cache** and return it

Example solution

```java
import java.util.*;

public class Playground {

    private static final Map<Long, Long> cache = new HashMap<>();

    private static long factorial(long number) {
        if (cache.containsKey(number)) {
            System.out.println("  Hit cache");
            // return immediately without calculating the result
            return cache.get(number);
        }

        long fact = 1;
        for(int i = 1; i <= number; i++) {
            System.out.println("  Multiplication");
            fact = fact * i;
        }
        // cache the result
        cache.put(number, fact);
        return fact;
    }
}
```

Example usage

```java
import java.util.*;

public class Playground {

    public static void main(String[] args) {
        System.out.println("Calculate factorial(5)");
        long r1 = factorial(5);
        System.out.println("Result: " + r1);
        System.out.println("Calculate factorial(10)”);
        long r2 = factorial(10);
        System.out.println("Result: " + r2);
        System.out.println("Calculate factorial(10)");
        long r3 = factorial(10);
        System.out.println("Result: " + r3);
    }

    //
    // Implementation of factorial
    //
}
```

Output

```
Calculate factorial(5)
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
Result: 120
Calculate factorial(10)
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
  Multiplication
Result: 3628800
Calculate factorial(10)
  Hit cache
Result: 3628800
```

Break

10 min

The lecture will continue at **16:55**

Outline

- Generics (part 1)
- Generics (part 2)
- Object data types
- **Error handling**

Runtime error

- If an error occurs during program execution, normal execution is aborted and an error object is created (thrown)
- The class **`Throwable`** captures all types of runtime errors
- An error object can be caught and handled appropriately
- There are 3 main categories of exceptional conditions

1. Checked exceptions: you are required to handle them
2. Unchecked exceptions / runtime exceptions: you are **not** required to handle them

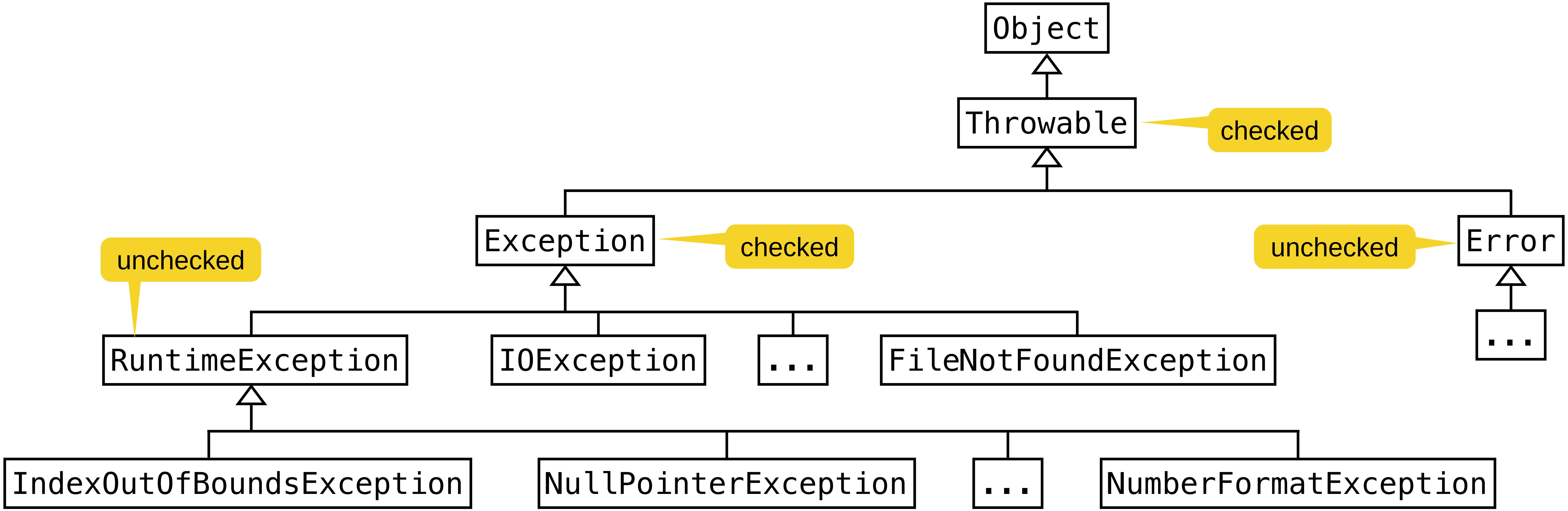
runtime and **unchecked** exceptions refer to the same: we can use them interchangeably

3. Errors: serious and usually irrecoverable conditions like a library incompatibility, infinite recursion, or memory leaks (also unchecked)

https://www.baeldung.com/java-exceptions

Error classes

- Explicit separation of
  - Normal program flow (which should be efficient and clear)
  - Handling of **special cases** (such as illegal inputs, incorrect use, security attacks, ...)



Throwable

- Subclasses
  - **`Error`**: represents severe problems that typically lead to program termination
  - **`Exception`**: covers recoverable conditions or disruptions
- For methods: Any uncaught **`Exception`** type must be declared in the method signature
- Example

```
public void readFile(String fileName) throws IOException {
    // Method code here
}
```

`RuntimeException`

- The subclass **`RuntimeException`** of the class **`Exception`** summarizes the exceptions that may occur during normal program execution
- A **`RuntimeException`** can occur at any time
- Therefore, it does not need to be declared in the header of the methods
- It can, but does not have to be **caught**
- **Types of error handling**
  - Ignore
  - Catch and treat where they occur
  - Catch and treat elsewhere

No error handling?

- If an error occurs and is not handled, the program execution aborts
- Example

```java
class Zero {
    public static void main(String[] args) {
        int x = 10;
        int y = 0;
        System.out.println(x / y);
    }
}
```

The program terminates because of division by **`(int)0`** and returns the error message

1. **Thread** (covered later) in which the error occurred

2. **Name** of the error class followed by the error message (defined by the **`getMessage()`** method)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:5)
```

3. **Stack trace:** the **function** in which the error occurred, more precisely the specification of all calls in the **method call stack**

Error handling

- If the program execution should **not** be terminated, the error **must** be caught
- Example: **`NumberFormatException`**

```
import java.util.Scanner;

public class Adding {
    public static void main(String[] args) {
        int x = getInt("1. Number: ");
        int y = getInt("2. Number: ");
        System.out.println("Sum: " + (x + y));
    }
    public static int getInt(String question) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            try {
                System.out.print(question);
                String input = scanner.next();
                return Integer.parseInt(input);
            }
            catch (NumberFormatException e) {
                System.out.println("Wrong input! ...");
            }
        }
    }
}
```

Read two integer values and sum them up

The handling of these errors is hidden in the function **`getInt()`**

During the input errors can occur, e.g. because a syntactically correct number is **not** entered

Example output

```
1. Number: e
Wrong input! ...
1. Number: 1
2. Number: v
Wrong input! ...
2. Number: 2
Sum: 3
```

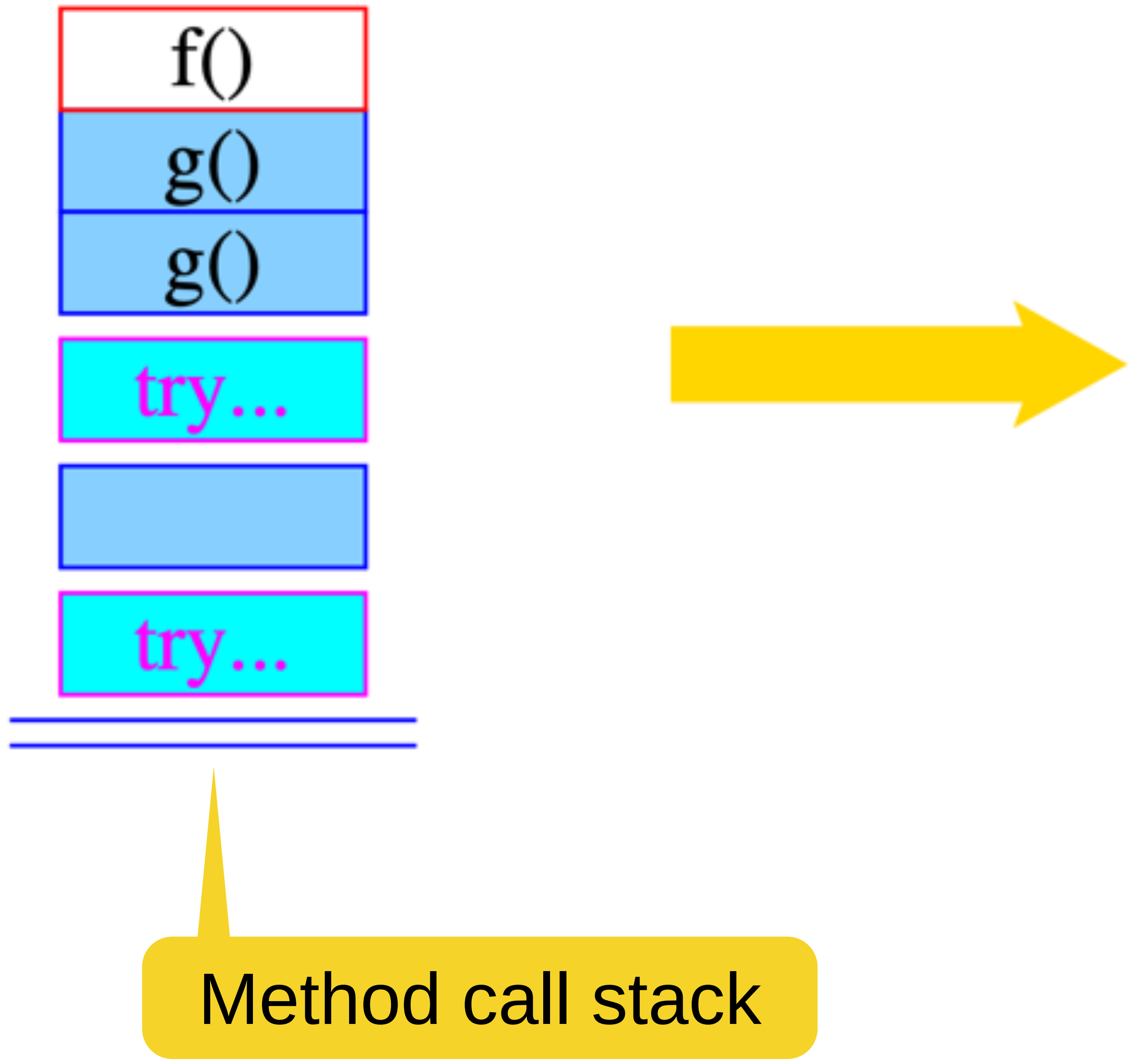
Exception handling

- An **exception handler** consists of a `try {...}` block where the error may occur followed by one or more `catch` rules
- If no error object is created when the statement sequence in the try block is executed, program execution continues directly after the handler
- If an exception is thrown, the handler sequentially searches the catch rules using the **thrown** error object

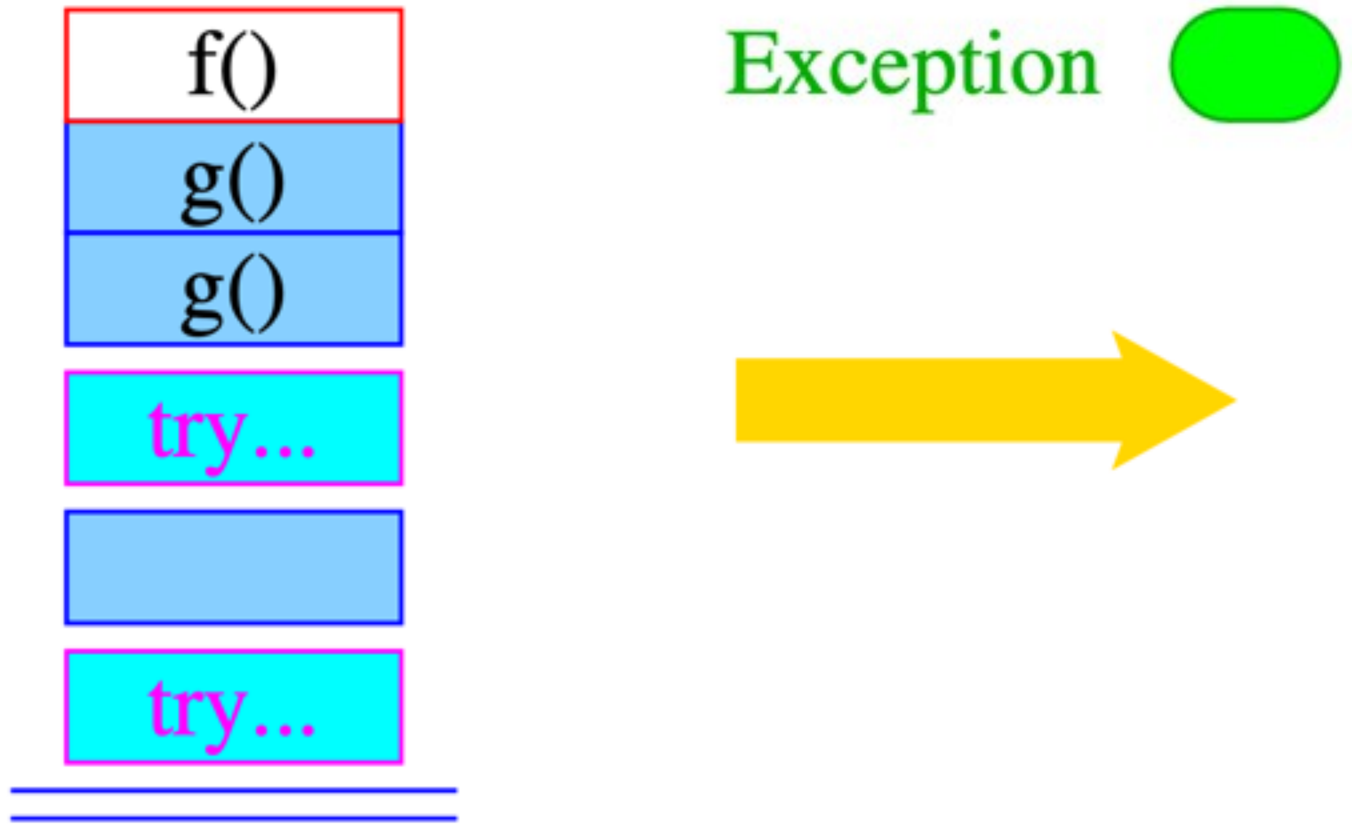
Exception handling

- Each catch rule is of the form: **`catch (Exc e) {...}`** where **`Exc`** specifies a class of errors and **`e`** is a formal parameter to which the error object is bound
- A rule is **applicable** if the error object is from (a subclass of) **`Exc`**
- The first catch rule that is applicable is applied
- Then the handler is exited
- If no catch rule is applicable, the error is propagated

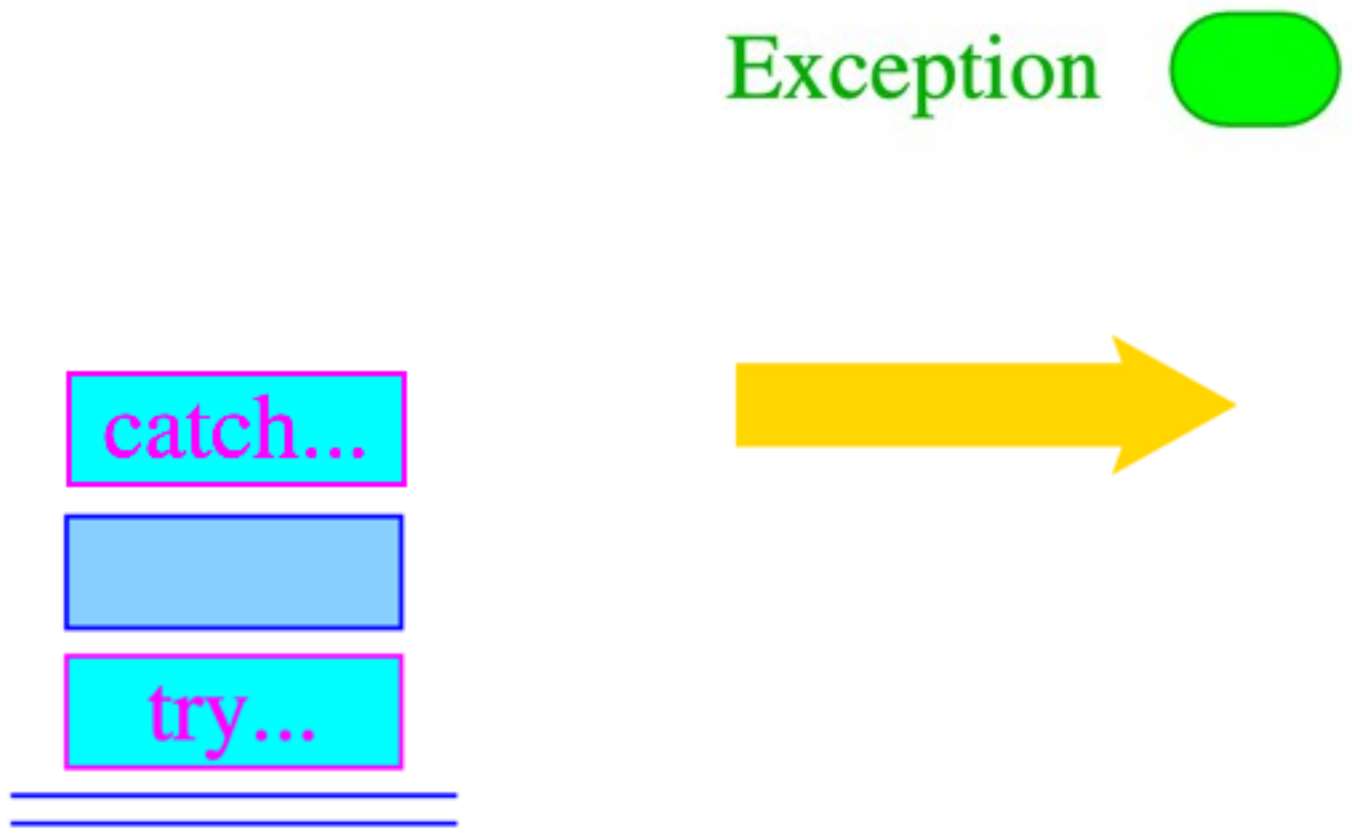
During runtime



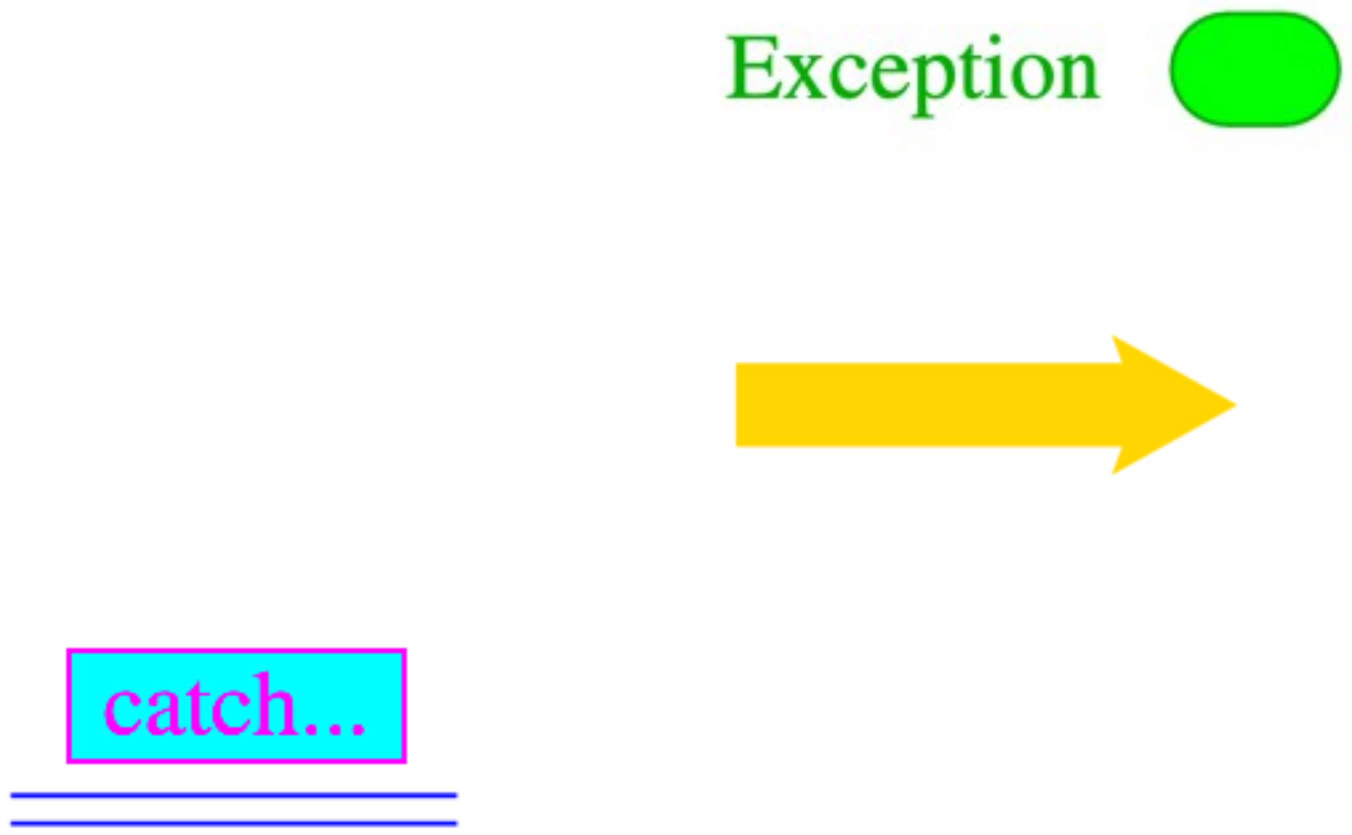
During runtime



During runtime



During runtime



During runtime

- Triggering an error abruptly leaves the current method call
- To keep the program in an orderly state despite the occurrence of the error, cleanup is often required - e.g. closing input / output streams   Covered later
- For this purpose **`finally {...}`** is used after a **`try`** statement

Final block

- The statements in the **`final`** block are executed in any case
- If no error is thrown in the try block, it is executed following the **`try`** block
- If an error is thrown and handled by a **`catch`** rule, it is executed following the block of the catch rule
- If the error is not handled by any **`catch`** rule, the final block is executed and then the error is passed on

Example NullPointerException

```java
public class Kill {

    public static void kill() {
        Object x = null;
        x.hashCode();
    }

    public static void main(String[] args) {
        try {
            kill();
        }
        catch (ClassCastException ex) {
            System.out.println("Wrong class!");
        }
        finally {
            System.out.println("Nothing was caught ...");
        }
    }
}
```

Code will be executed, but the program might still terminate

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "Object.hashCode()" because "x" is null
    at test.Kill.kill(Kill.java:7)
    at test.Kill.main(Kill.java:12)
Nothing was caught ...
```

Define exceptions

- You can also define custom exceptions and throw them
- Example

```java
class Killed extends Exception {
    Killed(String message) {
        super(message);
    }
}

public class Kill {
    public static void kill() throws Killed {
        throw new Killed("Killed the program");
    }
    public static void main(String[] args) {
        try {
            kill();
        }
        catch (RuntimeException rEx) {
            System.out.println("RunTimeException " + rEx + "\n");
        }
        catch (Killed bEx) {
            System.out.println("Killed It!");
            System.out.println(bEx);
            System.out.println(bEx.getMessage());
        }
    }
}
```

Throws the error of type **`Killed`** with a specific message

Output:

```
Killed It!
test.Killed: Killed the program
Killed the program
```

Customized errors

- A custom defined error **`XYZException`** should be declared as a subclass of **`Exception`**
- The class **`Exception`** has the constructors
  `public Exception();`
  `public Exception(String message);`
- **`throw new XYZException()`** throws the error - if the expression evaluates to an object of a subclass of **`Throwable`**
- **`Killed`** is not a subclass of **`RuntimeException`**, so the thrown exception is caught by the **second** catch rule

Errors

- Errors in Java are objects and can be handled by the program itself
- **`try ... catch ... finally`** allows to clearly separate error handling from normal program execution
- The predefined error types are often sufficient
- If special new errors/exceptions are needed, they can be organized in an inheritance hierarchy
- Exceptions **must not be used** to fix programming errors

Careful error handling

- Java's error mechanism should also only be used for error handling
  - Installing a handler is **cheap**; catching an **`Exception`** on the other hand is **expensive**
  - A normal program flow can be obfuscated to the point of opacity by using exceptions
  - What happens when in **`catch`** or **`finally`** blocks errors are thrown?
- Errors should be handled where they occur
- It is better to catch more **specific** errors than **general** ones
  - Avoid **`catch (Exception e) {...}`**

Exercise L05E05

10 min

- Create a custom error for validating the access for adults (age > 17) in the Age class
- Use the **`throw`** statement with an **`IllegalAccessException`** if the age is below 18
- Invoke the **`checkAge(...)`** method and handle the exception

```java
public class Age {
    static void checkAge(int age) {
        //TODO
    }
}
```

- **Optional challenge:** create your own **`Exception`** class **`Below18Exception`** and use it instead of the **`IllegalAccessException`**

Example solution

```java
public class Age {
    static void checkAge(int age) throws IllegalAccessException {
        if (age <= 17) {
            throw new IllegalAccessException("Access denied - You must be at least 18 years old.");
        }
        else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        }
        catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Example solution (with comments)

Define that the methods throws

throw the actual exception object

Handle the case that the exception is thrown

```
public class Age {
    static void checkAge(int age) throws IllegalAccessException {
        if (age <= 17) {
            throw new IllegalAccessException("Access denied - You must be at least 18 years old.");
        }
        else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        }
        catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Example solution (optional challenge)

```
public class Age {
    static void checkAge(int age) throws Below18Exception {
        if (age <= 17) {
            throw new Below18Exception("Access denied - You must be at least 18 years old.");
        }
        else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        }
        catch (Below18Exception e) {
            e.printStackTrace();
        }
    }
}

class Below18Exception extends Exception {
    Below18Exception(String message) {
        super(message);
    }
}
```

Define your own exception

Next steps

- **Tutor group** exercises
  - T05E01 - The Dark Pingu Rises
  - T05E02 - Exceptional Encryption
- **Homework** exercises
  - H05E01 - The Loadinator - Rise of the Generics
  - H05E02 - Boot Hard
- Read the following articles
  - https://www.geeksforgeeks.org/generics-in-java
  - https://www.javatpoint.com/exception-handling-in-java

→ Due until **Wednesday, December 6, 13:00**

Summary

- **Generics** allow to create reusable code for multiple types
- Generic types can be bound to other types which provides many opportunities to precisely define and implement code
- Common **object data types** in Java help to handle many situations
  - `String`s are powerful and used very often, in particular with user input
  - Collection types such as `List` and `Set` provide more flexibility and customizability than simple arrays
  - `Map`s allow to store simple key value pairs, but should not be overused (instead define actual objects)
- **Error handling** makes programs more robust and prevents them from stopping if something unexpected occurs

References

- https://www.crio.do/blog/string-methods-in-java
- https://www.baeldung.com/java-generics
- https://www.tutorialspoint.com/java/java_generics.htm
- https://www.geeksforgeeks.org/generics-in-java
- https://www.javatpoint.com/generics-in-java
- https://www.javatpoint.com/exception-handling-in-java
- https://www.baeldung.com/java-exceptions
- https://www.geeksforgeeks.org/exceptions-in-java
- https://www.javatpoint.com/try-catch-block
- https://www.javatpoint.com/wrapper-class-in-java
- https://www.geeksforgeeks.org/wrapper-classes-java