Summary Week 5

Type Casting

- Only possible on **compatible** data types

Implicit type casting happens implicitly without any action by the developer → no special syntax required

- Compiler throws an error on **potential information loss**

- Implicit cast is only possible if no information is lost
  → 2 can become 2.0 (from `int` to `double`)
  → 2.0 can **not implicitly** become 2 (from `double` to `int`)

Type Casting can **explicitly** be enforced →
target data type is placed in parenthesis in front of the identifier

```
(<<data type>>) <<identifier>>;
```

- Only possible between compatible data types

Cast Overview Primitive Data Types

TO		boolean	char	int	double
FROM					
boolean		-	No	No	No
char		No	-	Explicit Cast: interpreted as ASCII Values	Explicit Cast: interpreted as ASCII Values
int		No	Explicit Cast: Interpreted as ASCII Values	-	Implicit Cast
double		No	Explicit Cast: Interpreted as ASCII Values	Explicit Cast	-

- Every **object** can be casted to every data type above its **linear** inheritance hierarchy
- The type of the variable does not matter
  - We can only cast objects...
  - ...along their inheritance hierarchy.

Method Instead of Casting

```
double x = Double.parseDouble("1.234");
int y = Integer.parseInt("1234");
boolean b = Boolean.parseBoolean("true");
```

- Instead of casting
- Offered methods through Java Wrapper-Classes
- Example: Convert a String to a Double (e.g. "1.234")
- Does not work for Strings, that are not numeric
- → int z = Integer.parseInt("Ich fliege"); throws Error
- → NumberFormatException

Division by Zero

- Division of any **integer** number by (int) 0.
  System.out.println(**1/0**);, when dividing two integer numbers by zero, ...-

  → a `java.lang.ArithmeticException` is thrown.
- Division of any number by a floating-point (**double/float**) 0.0
  System.out.println(**1.0/0.0**);, when dividing a number by a floating-point zero, ...

  → `Infinity`.
- Division of **0/0**

  → `NaN`.

Object Data Types

- ☑ Object data types store references to objects.
- ☑ Primitive data types store values directly.
- ☑ Each class provides an object data type.

- Each class inherits these two methods from `Object`
  - `toString()`
    - Returns a String representation of the object
    - Default: *ClassName@HashCode*
    - Return value should be **individualized** to the respective class
  - `equals(Object obj)`
    - Compares two objects
    - Default: Compares the memory addresses of the objects
    - Should be overriden for own classes to test specific attributes

Wrapper Classes

- ☑ Wrapper classes offer data type-specific methods and constants.
- ☑ A suitable object data type exists for every primitive data type.

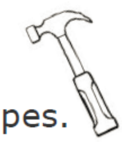
- Each primitve data type has a corresponding object data type
- They offer:
  - Methods specifically for that data type, e.g. **parsing**: `Integer.parseInt(String s)` to transform a String to an Integer
  - **Constants**, e.g. `Integer.MAX_VALUE` → 2147483647

Primitive Data Type	Corresponding Object Data Type
int	Integer
double	Double
boolean	Boolean
char	Character

Typical parsing method:

```
<<target data type>> parseTargetDT (<<source data type>> <<identifier>>){

  <<target data type>> result;

  [...] // logic to transform source variable
           to target data type goes here

  return result;

}
```

Parsing
Parsing transforms data types.
Java offers pre-defined parsing methods for all primitive data types.



Parsing is **not** casting. Parsing is done **instead** of casting.

Statics

Instantiation is for instances. And **static is for class level access**

- Constants should be public – so that everyone can benefit
- No one should be allowed to mess around with them
  - Make them `final`

  → They can not be changed any longer.

Static / Class Context
Static methods and variables can be accessed in class context.
They do not need an instantiated object.
Within static methods, the attributes and methods
of objects of this class are **not** accessible.
Static attributes are shared for all instances of the respective class.

Declare a variable in class context:

```
static <<data type>> <<identifier>>;
```

Declare a constant:

```
static final <<data type>> <<identifier>> = <<value>>;
```

Declare a method in class context:

```
static <<return type>> <<methodIdentifier>>(...){
  [...] // only local or static variables here
        // also: only call other static methods
}
```

Access a static variable / constant:

```
<<ClassIdentifier>>.<<variableIdentifier>>;
```

Call a static method:

```
<<ClassIdentifier>>.<<methodIdentifier>>(...);
```

- Correct usage:
  - Static attributes only for constants
  - Static methods only if they do not use the object's attributes

- In static methods, only static attributes and methods can be called

- Static methods can be called on an object of a corresponding class
  - Does not make any difference
    - same as stating the class name there

ArrayList

An ArrayList is a resizable array

- Once one array is full, a new array is automatically created
- Direct access to elements at any position

- Adding an element in between is more compute-intensive
  - Following elements will automatically be shifted

Define an ArrayList:

```
ArrayList< <<data type>> > <<identifier>>
                    = new ArrayList< <<data type>> >();
```


```
ArrayList<Vehicle> cars = new ArrayList<Car>();
```

Angle brackets! Supertype

Angle brackets! Subtype

Same data type or...

Proper methods of an ArrayList:

- [x] To insert a value at position x: `myAL.add(x, "myValue");`

- [x] To get the number of elements of an ArrayList: `myAL.size();`

Collection

- What are collections?
  - They **contain elements**
  - They may keep them in a certain **order**
  - They **offer access** to the elements
    - Efficiently
    - Based on specific characteristics (position, key)
  - Do not represent something themselves, they are a meta-structure

- Collections is an umbrella term, for specific structures

A collection represents a group of objects.
Objects inside a collection are called its elements.

- All collections allow to
  - Add elements (add)
  - Remove elements (remove)
  - Check whether an element is already in there (contains)
  - Retrieve the actual amount of elements in there (size)
  - Check whether they are empty (isEmpty)

- Specific structures are suited for specific use cases

Specific Lists:

- Queue: First In First Out (FIFO)

- Stack: Last In First Out (LIFO)

Set

- Each element can only be contained **once** → no duplicates
- Different implementations available in Java:
  - HashSet: unsorted
  - SortedSet: sorted

Map

- Every value has a distinct key assigned (KEY-VALUE pair)
  - Each key allows random access to its value
- Usually, keys are non-numeric (e.g. Strings, other objects, ...)
- Elements are unsorted (as opposed to lists)

Collections within the Java API

Hasso Plattner Institut

- Collections are almost always "better" than Arrays
- House-keeping is done implicitly (and correctly ☺)
- Collections only store object data types
  - Primitive data types → Wrapper classes are automatically used

Collections Overview

Type	Use Case	Implementation
List	random access via continous, numeric keys	ArrayList, LinkedList
Set	prevent duplicates (or implicitly remove them)	HashSet, SortedSet
Map	retrieve values via keys (key-value pairs)	HashMap

LIST:

- ADD: method to add new elements
- GETFIRST

- REMOVEFIRST

HASHMAP:

- PUT: method to add new elements
- GET(key): Retrieve value with key
- REMOVE(key)
- CONTAINSKEY(key): Check whether key exists (returns a boolean)

Some More Details on equals (and hashCode)

- hashCode is used to probe, whether equals is necessary
- contains methods work on hashCodes, as they are way faster than actually comparing full objects with equals
- Whenever equals returns true for two objects, their hashCodes **must** match
- Whenever you override equals(), you should override hashCode()
- Whenever you use a HashSet, a HashMap or other hash-based Collections, make sure that the hashCode does not change while they are in the collection.

How to implement hashCode()

- The method signature is: `public int hashCode()`
- Return value is an `int`, no parameters

- Rule of thumb to build a hash:
  - use existing numeric values (cast to `int`)
  - multiply them with prime numbers
  - add everything up
    - e.g.: `return age * 7 + shoesize * 13;`

- For complicated issues: there are libraries that greatly help
  - `Apache Commons Lang: HashCodeBuilder and EqualsBuilder`

Foreach Loops

- For each *something* in the *collection of somethings*, do …
- Can be used with arrays and collections

```
for (<<data type>> <<identifier>>: <<collectionIdentifier>>){
  [...]
}
```

- foreach is read-only! Values can not be deleted or added

- foreach-loops can basically be used with all collections. However some collections require "workarounds".

```
HashMap<String, String> roboLympics = new HashMap<>();

for (String s : roboLympics.keySet() ) {
   //do sth.
}
```