Introduction to Programming

René Thiemann

Institute of Computer Science
University of Innsbruck

WS 2008/2009

Outline

Outline

Motivation

```
class Student {
  String name;
  String title;
  int matrNr;
  String toString() {
    return title+" "+name;
  }
  int getMatrNr() {...}
}
```

```
class Employee {
  String name;
  String title;
  int salary;
  String toString() {
    return title+" "+name;
  }
  int getSalary() {...}
}
```

- problem: duplicated code `name`, `title`, `toString()`
- solution: extract common parts in super-class ...
- ... and build two extensions of this class containing the specific features

Example

- super-class

```
class Person {
  String name;
  String title;
  String toString() {
    return title+" "+name;
  }
}
```

- two extensions of this class (sub-classes)

```
class Student extends Person {
  int matrNr;
  int getMatrNr() {...}
}
```

```
class Employee extends Person {
  int salary;
  int getSalary() {...}
}
```


Extension of Classes

- syntax: `class Name extends OtherClassName { ... }`

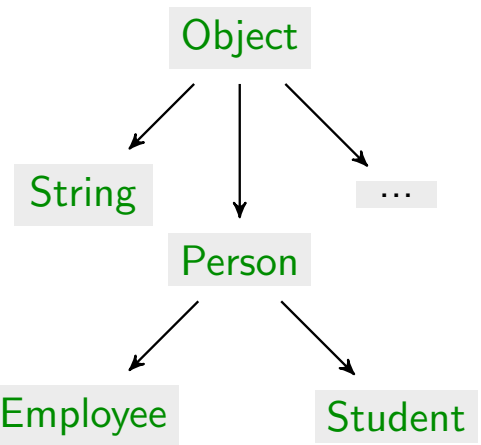
⇒ each class may have only one super-class

- shortcut: `class Name { ... }` is equivalent to

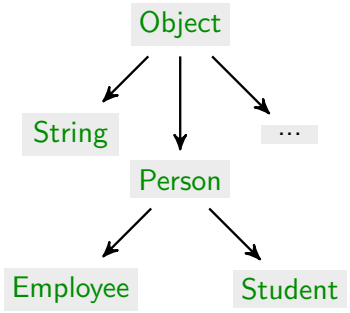
`class Name extends Object { ... }`

⇒ each class has exactly one super-class

⇒ exception: `Object` is the mother of all classes without super-class



Assignments



⇒ a Person is a Person, but also an Object

⇒ a Student is a Student, but also a Person and an Object

- variables of type Person can reference any person

⇒ assignment okay if variable type is higher than type of right-hand side

```
Person p = new Person();
Student s = new Student();
Object o = new Object();
p = p; // okay
p = s; // okay
o = s; // okay
s = p; // not okay
```


Casting

```
Person p;
p = s; /* okay */
```

```
Student s = new Student();
s = p; /* not okay */
```

- why is `s = p;` not okay? we know that p contains Student!
- but usually compiler cannot know:

```
Person p; Student s;
if (readInt() == 42)
      p = new Student();
else  p = new Person();
s = p; // not okay
```

- solution casting: `s = (Student) p;`
- you assert that object that is referenced by p is a Student

⇒ this will be checked during execution

  - if object is a Student everything is fine
  - if not, then program execution is aborted with "class cast exception"

Instanceof

```
Person p;                Student s = new Student();
p = s; /* okay */        s = (Student) p; /* okay */
```

- but what about the other program?

```
Person p; Student s;
if (readInt() == 42)
      p = new Student();
else  p = new Person();
s = (Student) p; // extremely dangerous
```

how to decide whether last assignment is possible?

- solution: use keyword instanceof: `if (p instanceof Student) { .. }`
- `expr instanceof ClassName` checks whether the expression is an object of a (sub-class of) ClassName

```
new Person() instanceof Person   // true
new Student() instanceof Person  // true
new Object() instanceof Person   // false
null instanceof Person           // false
```


Accessing Attributes and Calling Methods

- all attributes and methods of class can be used
- all attributes and methods of super-classes can be used
- important is type of variable/expression, not the type of object

```
class Person {
  String name; ...
  String toString() { ... }
}
class Student extends Person {
  int matrNr;
  int getMatrNr() {...}
}

Student s = new Student(); Person p = s;
s.matrNr = s.getMatrNr(); // okay
s.name   = s.toString();  // okay
p.name   = p.toString();  // okay
p.matrNr = p.getMatrNr(); // not okay
int x = ((Student) p).getMatrNr(); // okay
```

Access Modifiers of Attributes and Methods

- so far:
  - **public**: accessible from everywhere
  - no modifier: accessible only within package (not explained)
  - **private**: accessible only within class
- new:
  - **protected**: accessible only within class, package, and within subclasses

Overwriting of Methods and Hidden Attributes

```
class Person {
  String name; String title;
  String toString() { ... }
}
class Student extends Person {
  int matrNr;
  int getMatrNr() {...}
}
class Employee extends Person {
  String company;
  String getCompany() {...}
}
```

- so far: extended class has additional attributes and methods
- reality: desired/unavoidable to redefine attributes and methods
  - desired: employees should be displayed with company-name

  $\Rightarrow$ redefine toString()
  - unavoidable: extension of class where only byte-code is provided

  $\Rightarrow$ do not know internal names

Overwriting: Motivation

```
class Person {
  String name; String title;
  String toString() { return title + " " + name; }
}
class Employee extends Person {
  private String company;
}
```

- aim: output of employees in format "title name (company)"
- problem: cannot do this in class Person since there company is not accessible (more important: logically it belongs into Employee)
- solution: overwriting of toString():

```
class Employee extends Person {
  private String company;
  String toString() {
    return title + " " + name + " (" + company + ")";
  }
}
```


Overwriting: Which Method is Called?

```
class Person {
  String name; String title;
  String toString() { return title + " " + name; }
}
class Employee extends Person {
  String company;
  String toString() {
    return title + " " + name + " (" + company + ")";
  }
}
Employee e = new Employee();
e.title = "Mr"; e.name = "Jones"; e.company = "IFI";
println(e.toString()); // Mr Jones (IFI)
Person p = e;
println(p.toString()); // Mr Jones (IFI)
println(((Person) e).toString()); // Mr Jones (IFI)
```

- calling a method always uses method from class of referenced object, it is not determined by variable/expression type (dynamic binding)

Overwriting: Details

```
class A {
  typeA someMethod(type1 parName1a, ... ) { ... }
}
class B extends A {
  typeB someMethod(type1 parName1b, ... ) { ... }
}
```

- sub-classes can overwrite existing methods where the method-name is identical and the parameter-types are identical
- names of the parameters can be chosen freely
- return type must be identical or may be subclass of the old type
  - typeA = typeB = **int** — okay
  - typeA = Object, typeB = String — okay
  - typeA = String, typeB = Object — not okay!
- the access modifier must be identical or more relaxed in new method
  - **public** typeA f(), **public** typeB f() — okay
  - **protected** typeA f(), **public** typeB f() — okay
  - **public** typeA f(), **private** typeB f() — not okay!

Overwriting: **super**

```
class Person { ...
  String toString() { return title + " " + name; }
}
class Employee extends Person { ...
  String toString() {
    return title + " " + name + " (" + company + ")";
  }
}
```

- problem: code is duplicated (computation of the string including name and title), writing Employee needs internal knowledge of Person

⇒ code is not modular, Person cannot change its internals easily

- solution: use keyword **super** to call method of super-class

```
class Employee extends Person { ...
  String toString() {
    return super.toString() + " (" + company + ")";
  }
}
```

Constructors and **super**

```
class Person { String title, String name;
  Person() { }
  Person(String title, String name) {
    this.title = title; this.name = name;
  }
}
class Employee extends Person { String company;
  Employee(String company) {
    this.company = company;
  }
  Employee(String title, String name, String company) {
    super(title, name); // calls Person(title, name)
    this.company = company;
  }
}
```

- **super** can also be used in the first line of a constructor

⇒ the constructor of the super-class is called

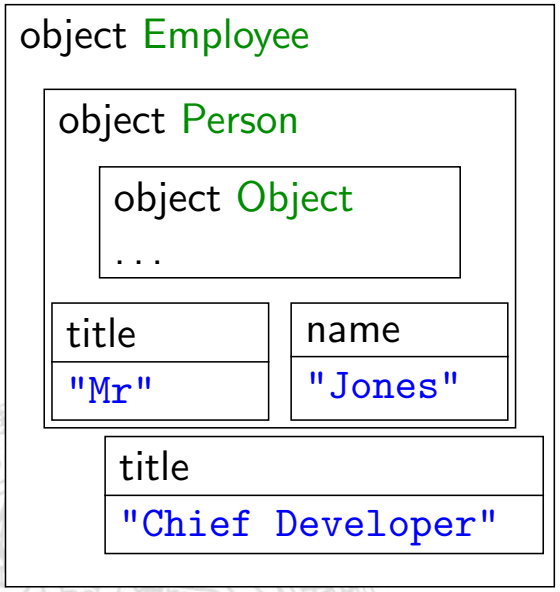
⇒ each class can initialize its elements, no code-duplication

- first line is not **this** (..) or **super**(..) ⇒ **super**() is inserted implicitly

Hidden Attributes

```
class Person {
  String title, String name; /* unknown! */
  public Person(String salutation, String surname) {
    title = salutation; name = surname;  /* unknown! */
  }
  public String toString() {
    return title + " " + name; /* unknown! */
  }
}
class Employee extends Person { String title;
  Employee(String salutation, String name, String title) {
    super(salutation, name); this.title = title;
  }
  public String toString() {
    return super.toString() + " (" + title + ")";
  }
}
new Employee("Mr", "Jones", "Chief Developer").toString();
```

Internal Structure of Objects



Hidden Attributes

```
class Person {
  String title; ...
  public String toString() {
    return title + ...; // A
  }
}
class Employee extends Person {
  String title; ...
  public String toString() {
    return super.toString() + " (" + title + ")"; // B
  }
}
```

- as for methods one can use the same name for attributes
- types can be arbitrarily chosen (**int** title is okay)
- in Employee the title of Person is hidden (but accessible via **super**)

⇒ the title in line B is the one of Employee

- access is determined statically by variable/expression and not by object that is referenced, different from method-calls

⇒ the title in line A is the one of Person

Hidden Attributes

```
class Person {
  String title; ...
  public String toString() {
    return title + ...; // A
  }
}
class Employee extends Person {
  String title; ...
  public String toString() {
    return super.toString() + " (" + title + ")"; // B
  }
}
Employee e = new Employee("Mr","Jones","Chief Developer");
println(e.title); // "Chief Developer"
println(((Person) e).title); // "Mr"
Person p = e;
println(p.title); // "Mr"
```


Comparing Attribute Access and Method Calls

- attribute access depends on expression/variable type
- method call depends on type of referenced object

```
class A {
  int x = 1;
  int f() {
    return 2;
  }
}
class B extends A {
  int x = 3;
  int f() {
    return 4;
  }
}

B b = new B();
A a = b;
println(b.x + " " + b.f()); // 3 4
println(a.x + " " + a.f()); // 1 4
```

Restricting Possibilities via **final**

- a method can be declared to be final

```
final int f() ...
```

⇒ it is not allowed to overwrite this method in subclasses

- a class can be declared to be final

```
final class A ...
```

⇒ it is not allowed to build subclasses of this class

⇒ implicitly all methods of that class are final

- an attribute can be declared to be final

```
final static double PI = 3.141592653589793;
```

⇒ it is not allowed to change the value of this attribute

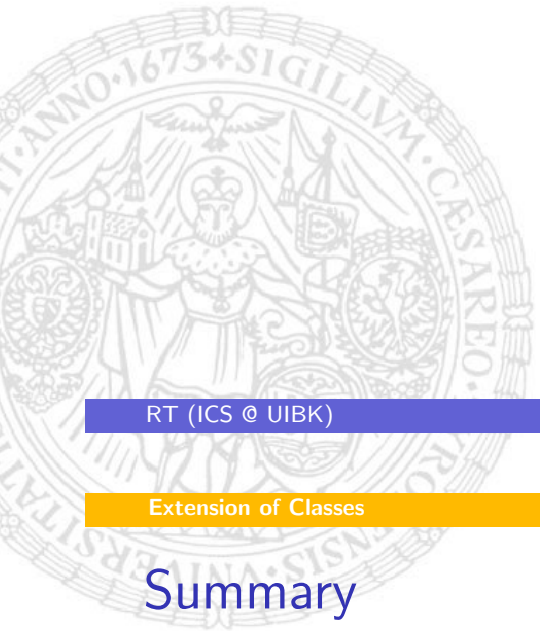
- static final attributes are often used to define constants
- non-static final attributes (e.g., id) must get value in the constructor

Guidelines

- define constants as `public static final type constName = value` where you might use another access modifier
- usually define attributes to be private: `private type name`
- if you have read-only attributes use `private final type name` or perhaps `public final type name`
- define internal methods as `private type name(..)` or `protected type name(..)` depending on whether internal methods can be used for sub-classes
- use the keyword **final** for methods only if the method must not be changed
- use the keyword **static** for methods whenever possible
- use the keyword **final** for classes only if there must be no subclasses

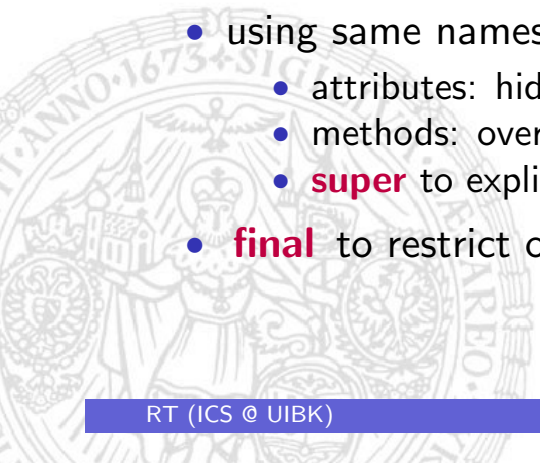
Example: Person

- class should be extendable, consists of name and id
- name can be changed due to marriage, ...
- id should be different for every person and cannot be changed



Summary

- class-extensions can be used to
  - reuse common functionality
  - have specific extensions
- extensions are done via **class** Subclass **extends** Superclass {...}
- variables of certain class can also reference objects of subclasses
- checking whether some object has type (subclass of) class: **instanceof**
- **protected** is new access modifier between **private** and **public**
- using same names in subclasses:
  - attributes: hiding, access via (static) type of expression
  - methods: overwriting, access via (dynamic) type of object
  - **super** to explicitly access attributes/methods of superclass
- **final** to restrict overwriting, extension of classes



Outline

Maps

- consider you want to use a dictionary (map) with the following features
  - storing an entry under a keyword ("tree" → "a plant with leafs ...")
  - removing an entry for some keyword
  - iterate through all keywords ("tree", "car", "Platon", ...)
  - combine two dictionaries
- setup classes

```
public class Map {
  public int size() {...}
  public boolean put(String key, String value) {...}
  public String get(String key) {...}
  public Iterator keys() {...}
  public boolean putAll(Map otherMap) {...}
  public String toString() {...}
}
public class Iterator {
  public String next() {...}
  public boolean hasNext() {...}
}
```

Using a Map

```
Map m = new Map(..);
m.put("tree", "a plant with ...");
m.put("car", "a vehicle ...");
Iterator i = m.keys();
while (i.hasNext()) {
  println("next key is " + i.next());
}
println("a tree is " + m.get("tree"));
```

- two possible outputs (both are okay)

```
next key is tree
next key is car
a tree is a plant with ...
```

```
next key is car
next key is tree
a tree is a plant with ...
```


Interfaces

- several possibilities to implement maps
  - efficient/inefficient
  - (un)bounded number of keys
  - order of iteration of keys (alphabetical, order of addition)
  - ...
- aims:
  - access all maps in a uniform way
  - easily exchange one map-implementation by another
- problems:
  - different implementations may use different names for same functionality, offer different features
  - all variables have to be changed from MapImpl1 to MapImpl2
- solution: use interfaces
  - an interface specifies desired functionality without implementing it
  - classes can then implement this interface
  - external code should rely on interface, never on concrete class
  - each interface is an own type where all classes fit which implement the interface (i.e., the implementing class is a "subclass of the interface")

Interfaces in Java

- instead of **public class** Name { ... } write

  public interface Name { ... }

- an interface may only contain public-method-declarations without implementations (";" instead of { ... })

```
public interface Map { /* documentation */
  public int size();
  public void put(String key, String value);
  public String get(String key);
  public Iterator keys();
  public void putAll(Map otherMap);
  public String toString();
}
public interface Iterator { /* documentation */
  public String next();
  public boolean hasNext();
}
```

- constants are also allowed: **public double** PI = 3.14;

Working with Interfaces

- previous code

```
Map m = new Map(..);
m.put("tree", "a plant with ...");
m.put("car", "a vehicle ...");
Iterator i = m.keys();
while (i.hasNext()) {
  println("next key is " + i.next());
}
println("a tree is " + m.get("tree"));
```

  remains nearly unchanged

- only problem: call of constructor **new** Map(..) :
  - it is never possible to create instance of interface (which implementation should be used?)
  - here one needs to have at least one class which implements the interface

  ⇒ interface types can be used as class types except that there are no constructors, i.e., no **new** InterfaceName(..) !

Implementing Interfaces

- each class can implement several interfaces

```
class Name
    extends SuperClass
    implements Interface1, Interface2, ... {
    ...
}
```

- when implementing an interface like Interface1 then
  - all methods within Interface1 must be implemented in class Name
  - except those methods that have already been implemented in class SuperClass

Example: Map Implementation Based On Arrays

```
public class ArrayMap implements Map {
    protected String[] keys;
    protected String[] values;
    protected int n;
    public ArrayMap(int maxSize) {
        keys = new String[maxSize];
        values = new String[maxSize];
        n = 0;
    }
    public int size() {
        return n;
    }
    protected int findIndex(String key) {
        for (int i=0; i<n; i++) {
            if (keys[i].equals(key)) {
                return i;
            }
        }
        return n;
    }
```

Example (continued)

```
    public String get(String key) {
        int i = findIndex(key);
        if (i == n) {
            return null;
        } else {
            return values[i];
        }
    }
    public boolean put(String key, String value) {
        int i = findIndex(key);
        if (i < keys.length) {
            keys[i] = key;
            values[i] = value;
        } else {
            return false;
        }
        if (i == n) {
            n++;
        }
        return true;
    }
```


Example (continued)

```
    public boolean putAll(Map other) {
        Iterator i = other.keys();
        boolean res = true;
        while (i.hasNext()) {
            String key = i.next();
            res = put(key, other.get(key)) && res;
        }
        return res;
    }
    public String toString() {
        String res = "(";
        for (int i=0; i<n; i++) {
            if (i > 0) {
                res += ", ";
            }
            res += keys[i] + " -> " + values[i];
        }
        return res + ")";
    }
```

Example (continued)

```
    public Iterator keys() {
        return new ArrayIterator(keys,n);
    }
}
```

```
class ArrayIterator implements Iterator {
    String[] a;
    int n;
    ArrayIterator(String[] array, int n) {
        this.a = array;
        this.n = n;
    }
    public boolean hasNext() {
        return n > 0;
    }
    public String next() {
        n--;
        return a[n];
    }
}
```


Extension of Interfaces

- interfaces can also be extended

```
public interface Name extends Interface1, In2, ... {
    ... some method declarations ...
}
```

- when implementing Name ...
  - all methods that are specified directly within Name must be implemented
  - all methods of Interface1, In2, ... must be implemented
- example:

```
public interface ExtendedMap extends Map {
  public void remove(String key);
}
```

Aim: Reuse Code

- so far: each class implementing an interface must implement all methods of the interface

⇒ large interfaces require lots of work for implementations

- observation: several methods M may be similar for each implementation

⇒ idea: implement methods M only once

⇒ solution:
  - implement methods M in super-class without implementing other methods of interface
  - build different sub-classes which implement the other methods which depend on concrete realization

Abstract Classes

- observation: in interface Map, methods putAll(), toString(), size() can be implemented without knowing internals

⇒ create super-class AbstractMap which only implements these methods without implementing put(), get(), and keys()

- problem: how to express that methods like put() are not implemented
- solution: use abstract class with abstract methods
- abstract method: method without implementation (as in interfaces)

```
abstract type someMethod(type1 name1, ...);
```

- abstract class: class containing at least one abstract method must be declared abstract

```
abstract class SomeClass { ... }
```

abstract classes can be used like normal classes,
except that it is not allowed to create instances

⇒ no **new** SomeClass(...)

Example: Abstract Map

```
public abstract class AbstractMap implements Map {
    public int size() {
        int n = 0;
        Iterator i = this.keys();
        while (i.hasNext()) {
            i.next();
            n++;
        }
        return n;
    }
    public boolean putAll(Map other) {
        Iterator i = other.keys();
        boolean res = true;
        while (i.hasNext()) {
            String key = i.next();
            res = this.put(key, other.get(key)) && res;
        }
        return res;
    }
```


Example: Abstract Map (continued)

```
    public String toString() {
        String res = "(";
        boolean first = true;
        Iterator i = this.keys();
        while (i.hasNext()) {
            String key = i.next();
            if (first) {
                first = false;
            } else {
                res += ", ";
            }
            res += key + " -> " + this.get(key);
        }
        return res + ")";
    }
    public abstract String get(String key);
    public abstract boolean put(String key, String value);
    // public abstract Iterator keys();
}
```

Example: ArrayMap based on AbstractMap

```
public ArrayMap2 extends AbstractMap {
  // no "implements Map" required

  // attributes keys, values, n as in ArrayMap

  // constructor as in ArrayMap

  // methods findIndex(), get(), put(), keys() as in ArrayMap

  // methods putAll(), toString(), size() not required!
  // nevertheless own implementation of size() for efficiency
  public int size() {
      return n;
  }
}
```


Summary

- interfaces should be used by external code instead of concrete classes

⇒ easy to change implementation

- abstract classes are classes where not all methods have been implemented

⇒ can be used to implement large parts of interfaces

⇒ only specific parts have to be implemented for concrete classes

- interfaces and abstract classes are similar but they are not the same:
  - both cannot be instantiated, i.e., no `new Inferface_AbstractClass (..)`
  - both can contain abstract methods without implementation
  - abstract classes consist of implemented and unimplemented methods
    interfaces has only unimplemented methods
  - a class can be an extension of only one abstract class
    but can implement many interfaces

Outline

Type of Data in Collection Classes

- recall type hierarchy: everything is an Object
- if data of list, map, stack, set,... is Object then these collection classes can be filled with any kind of objects, i.e., Persons, Strings, ...

```
public interface Map {
  public boolean put(Object key, Object value);
  public Object get(Object key);
  ...
}

Map m = new ArrayMap(5);
m.put("tree", "a plant ...");
m.put("car", "a vehicle ...");
String s = (String) m.get("tree"); // cast required
```

- two problems
  - primitive datatypes (**int**,**boolean**,...) are no objects
  - map needs equality; how can this method be called?

Primitive Datatypes as Data for Collection Classes

- collection classes only accept objects: `public void insert(Object data)`

⇒ `int x = 5; List l = new List(); l.insert(x)` is problematic

- solution: for each primitive datatype there is a wrapper class
  - Integer, Double, Boolean, ...
  - these classes store primitive data internally
  - and provide additional functionality (only Integer as example):
    - `public static Integer valueOf(int x)`: generates Integer from int
    - `public int intValue()`: returns internal int-value
    - `public static int parseInt(String x)`: transforms string to integer
    - `public final static int MAX_VALUE`: largest possible int-value

⇒ `l.insert(Integer.valueOf(x))` and

`x = ((Integer) l.removeHead()).intValue()` are okay

- since Java 1.5 auto boxing is available: it automatically converts primitive data to object of wrapper class and vice versa: `l.insert(x)` and `x = (Integer) l.removeHead()` is automatically expanded to code above

Equality for Collection Classes

- certain collection classes need to know whether objects are equal
  - Set: collection of elements without duplicates
  - Map: mapping from keys to values, e.g., persons to addresses

```
public void put(Object key, Object value) {...}
Map m = ...;
m.put(new Person("John","Wood","3D023"), "Arzl");
m.put(new Person("John","Wood","4X738"), "Rum");
```

⇒ new address or different "John Wood"s? when are two persons equal?

- problem: `==` only checks for same references (in above code the John Woods would be different), inflexible
- solution: class Object defines method

```
public boolean equals(Object other) {...}
```

⇒ whenever one has to compare objects, use `equals`, but not `==`

⇒ collection classes use `equals` to decide equality

⇒ each class that is used for comparisons (keys, data for sets, etc.) should overwrite equals-method and define its own equality

Example

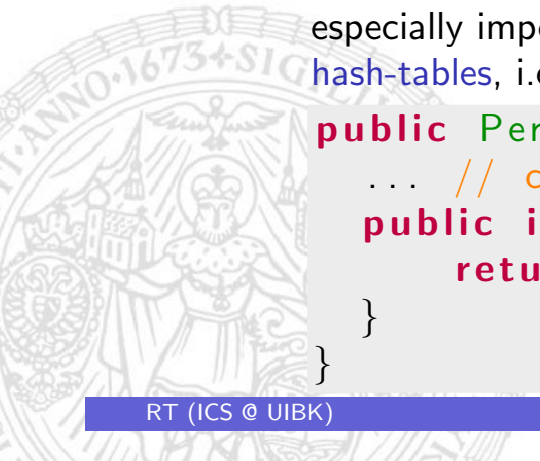
The Class Object

- since Object is super-class of all classes, its methods are available for every class

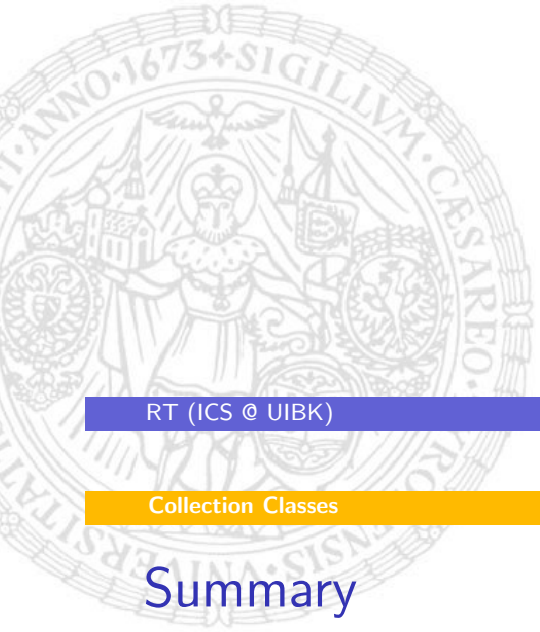
⇒ these methods should be overwritten to reflect the properties of specific class

- methods:
  - **public** String toString(): displaying an object
  - **public boolean** equals(Object o): comparison with other objects
  - **public int** hashCode(): should be overwritten whenever equals is overwritten
    only requirement: two "equal" objects must return same hashcode; especially important for collection classes which are based on hash-tables, i.e., HashMap, HashSet, LinkedHashMap, ...

```
public Person {
   ... // code from previous slide
   public int hashCode() {
       return socialSecurityID.hashCode();
   }
}
```



The Library java.util



Summary

- collection classes like lists are already implemented in libraries
- usually these libraries contain interfaces, abstract classes, and concrete classes similar to map-example of previous section
- data type is always Object

$\Rightarrow$ collections can store every kind of data

- for primitive data types use wrapper classes
- whenever comparisons are essential, implement equals-method