Create Your Own Universe

- Classes and Behavior
- Types and Objects

Review Questions

- What is the difference between a function and a void method?
- What does the return statement do?
- What is the meaning of **`private`**?
- How do you determine the type of a statement?
- Can there be, in a single program :
  - two classes with the same name?
  - two methods or attributes with the same name?
  - Two methods with the same name within the same class?

Method Overloading

- Methods are identified by their signature.
- I.e. two methods with the same name can coexist, if their parameter lists differ (in number or type)
- This is called **method overloading.**

- Oberloading is forbidden, if method calls are indistinguishable to the compiler,
  - if their signatures only differ in the return type or access modifier
  - if they can be matched through implicit parameter type casting
  - if they only differ in their static modifier.

Test: Overloading

Which methods can coexist?

```
1.  public int m(int a, String b);
2.  public void m(int a);
3.  public static void m(float a);
4.  public int m(int a, boolean b);
5.  private void m(boolean b);
6.  private void m(long a);
```

Using Overloading

```
public class BusinessCards
{   // standard version
    public static void printCards
        (String name, String address,
         int number)
    { ... }
    // more options
    public static void printCards
        (String title, String name,
         String company,
         String addres,
         int layout, int number)
    { ... }
}
```

I. Classes and Behavior

Method Specification

- All you need to know to call a method correctly is found in the **method header**:

```
public static int meth(char c)
```

- It is also called **method signature** (or footprint)
- Syntactically, a signature is a complete **method specification**.

- It does not tell you anything about the semantics → add comments!

Class Specification

- The information needed to use a class properly is called its **export interface,**
- described by the **class specification.**
- It consist of all **public declarations**:
  - Attributes: type and name
  - Methods: signatures
  - Comments explaining the samantics

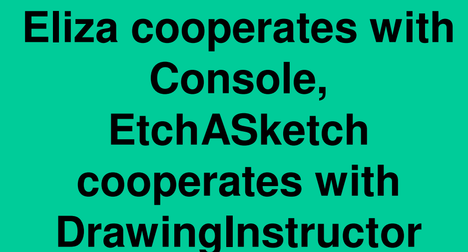
Classes "behave" and categorize

- To use a class means to call ist methods.
- Console "behaves":
  - if you call println, it will do output,
  - if you call readln, it will accept input and make it available to the caller.
- Console groups i/o behaviour
- Would you put a method called calculateMedian into Console?
  - No – it does not "fit".
  - Console methods deal with input and output.

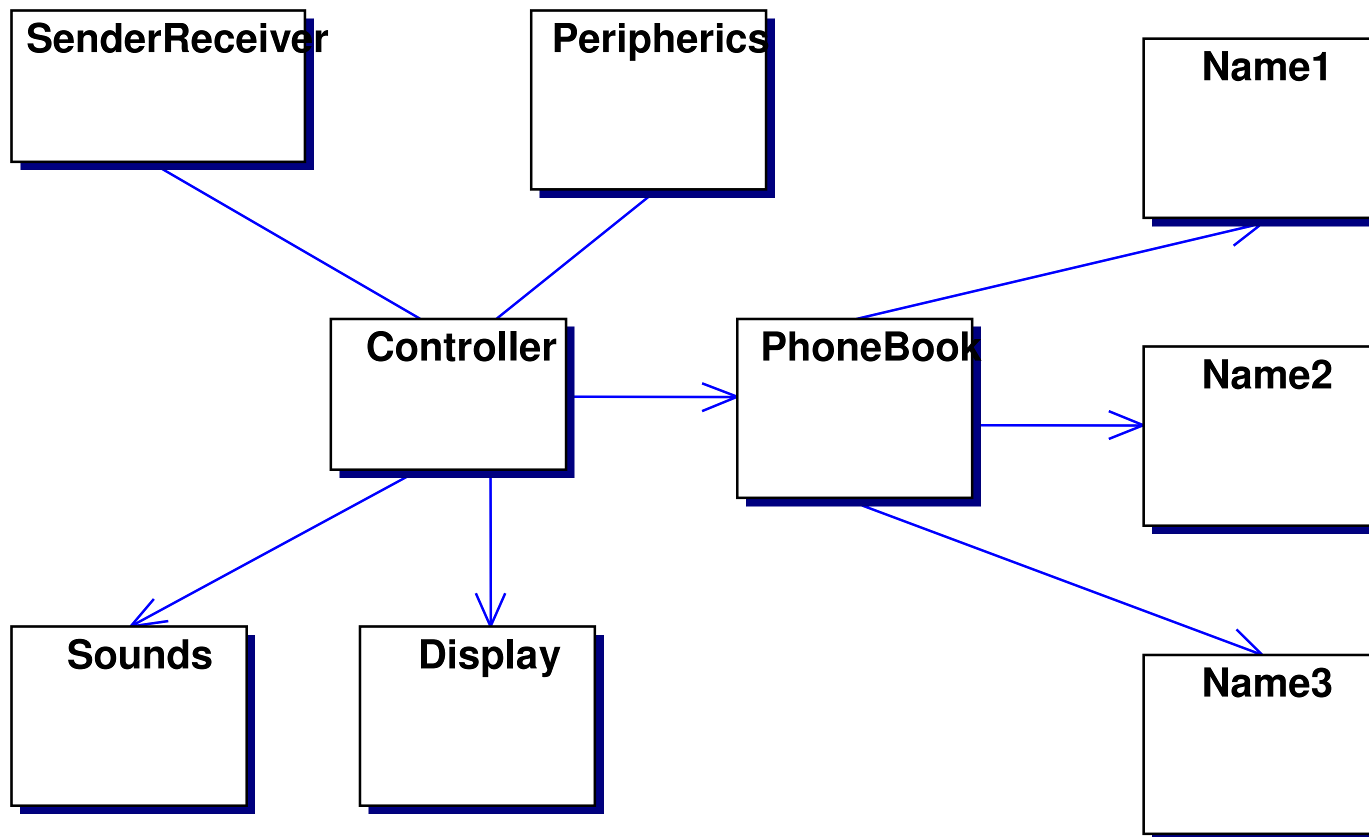
Objects

- Objects in real life
  - contain information of a certain category
  - behave (work, react)
  - coorperate or interact

- Classes in a Program
  - store information
  - behave according to their methods
  - cooperate



Cell Phone:
System of Interacting Units



Controller (incoming call)

state: free / busy

rules of behaviour:

- incoming call notification (from Sender/Receiver)
  - → busy
  - - lookup name and personalized ringing tone
  - - display name
  - - sound ringing tone
- call rejected (from Peripherals)
  - - stop ringing
  - - display neutral
  - → free
- call accepted (from Peripherals)
  - - open connection
  - - display name and time lapse
  - - open mic and phones
- call finished (from Peripherals)
  - - close connection
  - - close mic and phones
  - - display neutral
  - → free

Display

state: current text, time lapse

rules of behaviour:
- ➢display call
  - clear display
  - display required text / screen

Phonebook

state: current entry

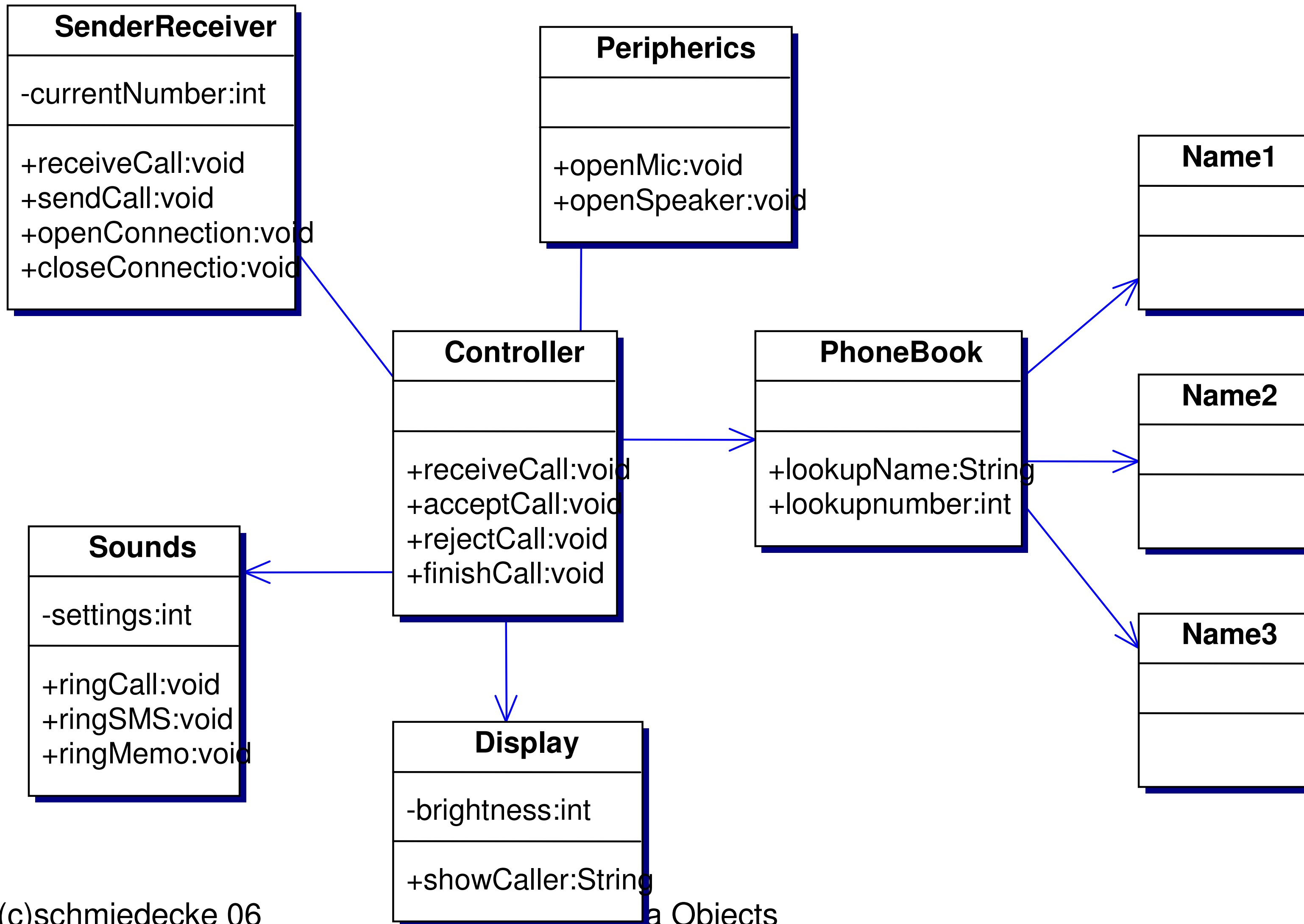
rules of behaviour:
- ➢lookup by number
  - return name and ringing sound
  - return null, if not found
- ➢lookup by name
  - -return number
  - return null, if not found

Sound

state: active / quiet, current sound

...

Cell Phone (Recieving call)



information

behavior

```
public class SenderReceiver {
  int currentCall = -100;
  public static void receiveCall(int number) {
    currentCall = number;
    Controller.receiveCall(number);
  }
  public static void sendCall(int number);
  public static void endCall();
  // ...
}
```

behavior

```
public class Controller {
  public static void receiveCall(int number) {
     Sounds.ringCall();
     String name = PhoneBook.lookupName(number);
     Display.showName(name);
  }
  public static void acceptCall();
  public static void rejectCall();
  public static void endCall();

  // ...
}
```

Programming interacting units

- I can program the Class Controller, if I have the specification of the classes Sound, Display, PhoneBook

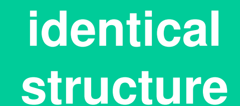
- In a System, each object contributes its behavior a the system, maybe using other objects' behavior.
- So a class can be considered an object...

II. Types and Objects

...Flaw in Concept

- Concept Class as Object credible → except for Name
- Just one class Name
- But we want many names in our phone book:

```
public class Name1 {
     static String name = "Helen";
     static int homeNumber = 007;
     static int cellNumber = 0815;
     editNumber();
}
public class Name2 {
     static String name = "Maurice";
     static int homeNumber = 2050;
     static int cellNumber = 0123;
     editNumber();
}
```



- *Problem1: program length ....*
- *Problem2: new entries*

Real Life Objects

- There are unique objects:
  - the Eiffel tower
  - the US president
  - the Control unit of your cell phone (within the system)
  - YOU...
- and there are many Objects that are instances of a type:
  - a lab workstation
  - a bicycle
  - a department
  - a phone book entry

Classes as Types

- Consider the unique object as a special case.
- Don't consider classes objects
- but as object blue prints,
  i.e. as **types:**

- A type defines
  - which information a "thing" can store → Attributes
  - and what can be done with this information
    → Methods

Class "Name" as Type

- A type defines
  - which information a "thing" can store → Attributes
  - and what can be done with this information → Methods
  - *another way of putting it: State Domain and Behavior Pattern*

Example: **Type Name**

State: name, home number, cell number, office number, group, ringing tone.

Behavior: edit name, edit number, add number, set group, set ringing tone, get home number, get cell number, ...

A Name Object (or "Name Instance")

An Object is an Entity which

- exists at runtime
- has a type
- possesses a *specific state* and
- shows a *specific behavior*

Example: **Name Object IlseSchmiedecke**

State: name = "ilse schmiedecke“
home number = 001-501-123456
cell number = 001 -251-665588
group = 24
ringing tone = emergency
Behavior: set group, get ringing tone…..

"Getter and Setter Methods"

- The state of an object is given by ist attribute values.
- Avoid making an attribute public.
- Define a set of functions to read the attributes - Getter:

```
public String getName();
public int getCellNumber();
```

- Define a set of void methods to change the attribute values - Setter:

```
public void setName(String name);
public void setCellNumber(int number);
```

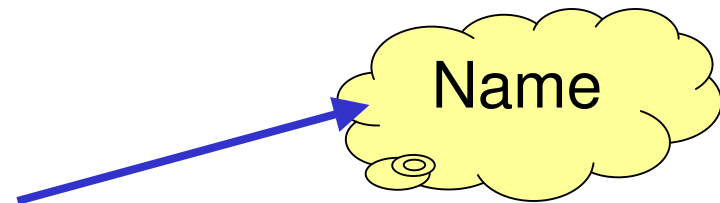
Creating Objects from Classes

- Type defined as a class:
- `public class Name { .... }`

- Object Instantiation: operator new

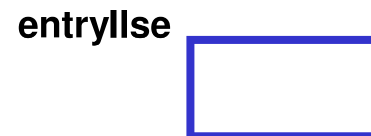
```
new Name()
```

  – creates a new object of type Name *somewhere in memory*
  – evaluating the new-expression yields a reference to the new object



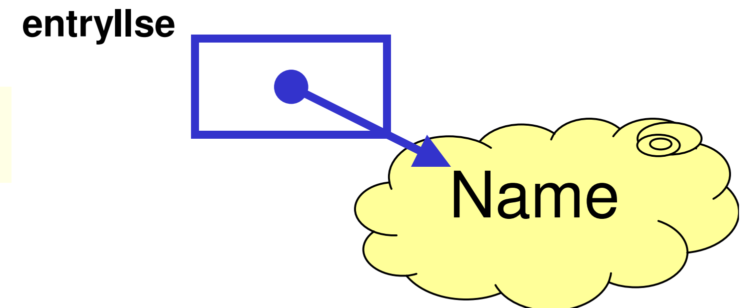
- Variable of Class Type

```
Name entryIlse;
```



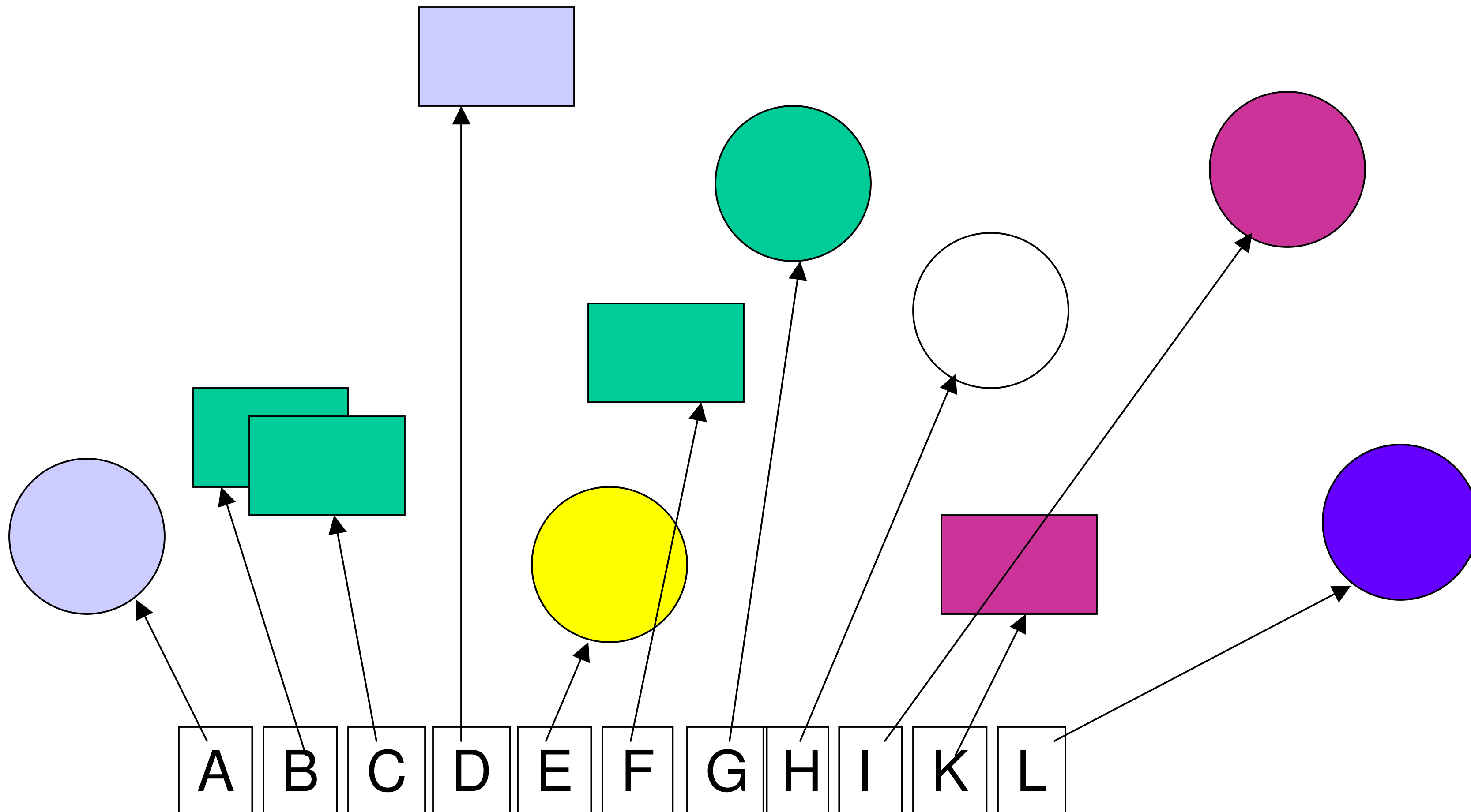
  - can store an object reference of given type:

```
entryIlse = new Name();
```



Objects and Variables

`new` can be used to create new Objects



Variables attach identifiers to them, so that they can be used.

Using different Objects

```
public class PhoneBook{
   public static void main(String[] args) {
      Name entryIlse = new Name();
      Name entryUwe = new Name();

      entryIlse.setRingingTone(emergency);
      entryUwe.setRingingtone(badinerie);
   }
}
```

not:
`Name.setRingingTone()`

- Variable (object reference) instead of class name
  action affects only one object of the class
- **`static`:**
  Attribute or method belongs to the class,
  is shared by all objects of the class,
  can even be used without any object (class name)

⇒ stop using **`static`** from now on!

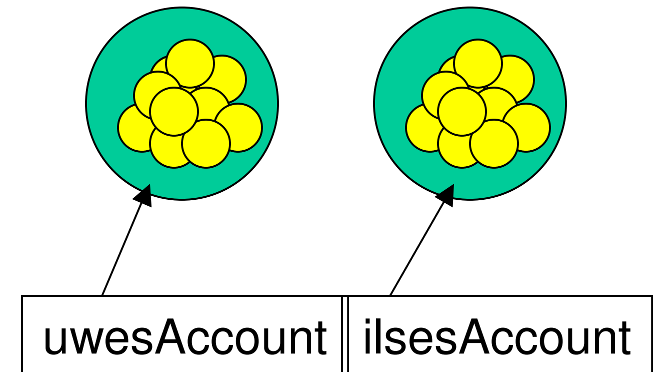
Static and Instance class Members

```
class Name {
  static int ownNumber;       // static, same for all Name objects
  static int defaultTone;
  static int defaultGroup;


  String lastName;            // object specific
  int homeNumber;             // or "instance specific"
  int officenumer;
  int cellNumber;
  String email;

  public static void setOwnNumber(int number);
  public static void setDefaultGroup(int group);
  public static void setDefaultTone(int tone);

  public int getHomeNumber();
  public void setHomeNumber();
  public int getOfficeNumber();
  public void setOfficeNumber();
  //...
}
```

Using Object Variables

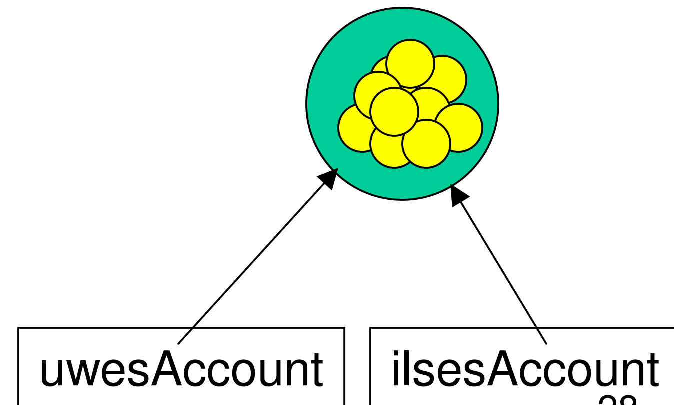
- **If two variables refer to different objects, their actions do not interfere:**

```
Account uwesAccount = new Account();
Account ilsesAccount = new Account();

ilsesAccount.withdraw(1000.00);
uwesAccount.withdraw(1000.00);
```



- **You can assign a variables to a variable, then they refer both to the same object:**

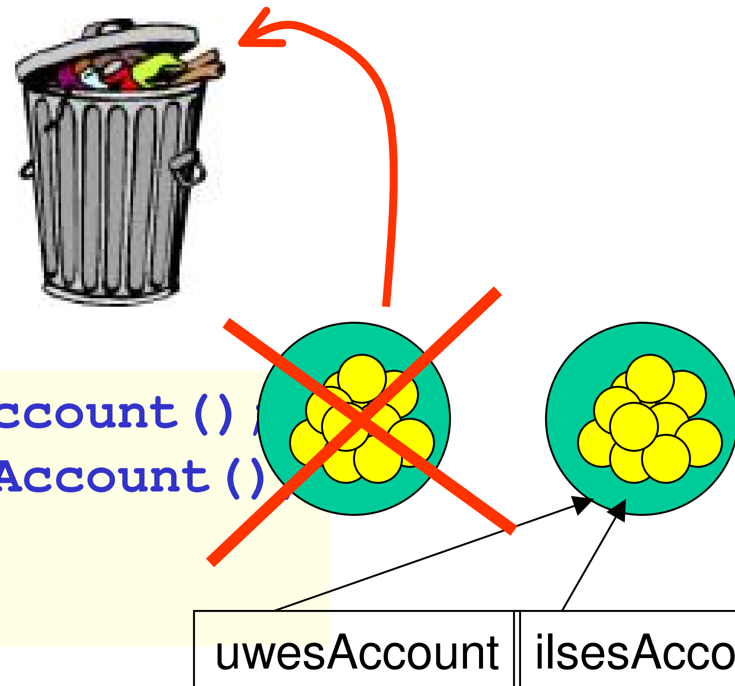
```
uwesAccount = ilsesAccount;

ilsesAccount.withhdraw(1000.00);
uwesAccount.withdraw(1000.00);
```



Loss of Objects

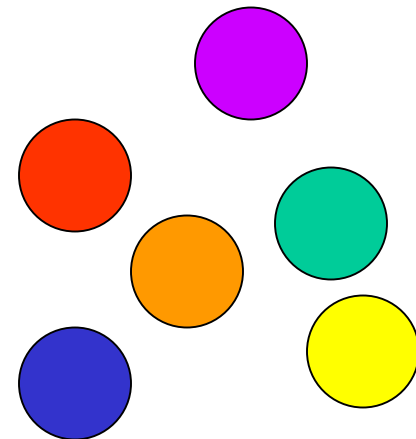
- Generally, objects live from their creation to the end of the program.
- If there is no reference to an object, it is lost (unreachable)
- In Java, a Garbage Collector collects these lost objects (at some time) and restores the storage space.

```
Account uwesAccount = new Account();
Account ilsesAccount = new Account();
uwesAccount = ilsesAccount;
// the old object is lost!
```



Juggling with Objects

```
Ball annaLeft, annaRight,
     timLeft, timRight;
annaLeft = new Ball("red");
annaRight = new Ball("blue");
timLeft = annaRight;
annaLeft = new Ball("yellow");
timRight = new Ball("green");
annaRight = annaLeft;
timLeft = new Ball("orange");
annaLeft = new Ball("purple");
```



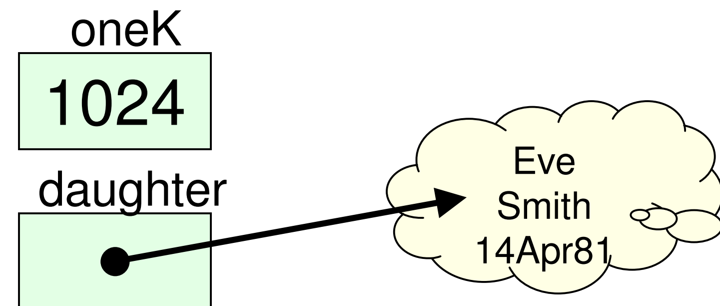
Who holds which ball? Which ones are lost?

Summary: Types

- Object types define object patterns.
- Variables of object types store references to objects.
- Using an object reference, you can call the object's methods.
- An object can have different states depending on ist attribute values.

- Primitive types have a direct representation on the machine– e.g. numbers.
- Values of primitve types are elements of a fixed set of values.
  They have no states.
- Variables of primitive types store values directly, not references to values.
- They cannot be used for method calls (but used with operators instead).

```
int oneK = 1024;
oneK.double() // error!

Person daughter = eve;
```



Some Java Standard Types

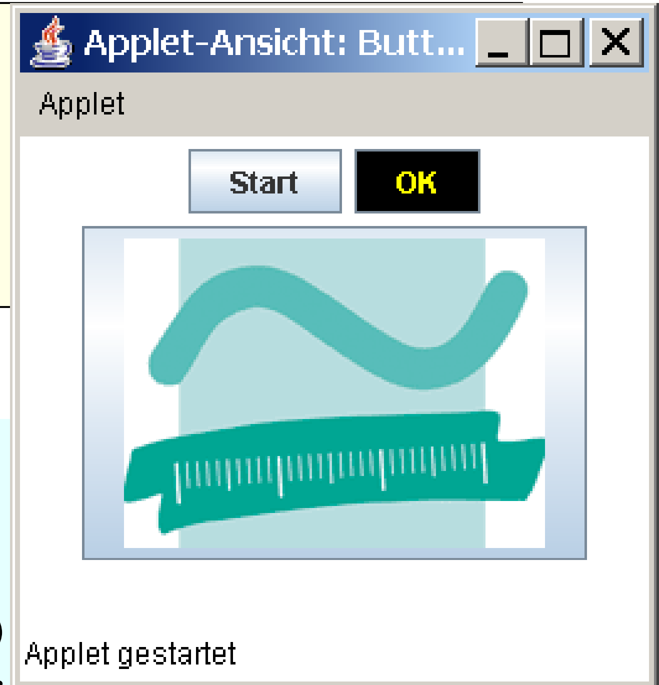
- `String`:
  - capitalize, get length, concatenate ...
- `DialogBox`
  - define text, show box, ...
- `JButton`
  - define text, change color, add action
- `Vector`
  - add object, find object, ...

....

Type JButton

```
JButton button1 = new JButton();
JButton button2 = new JButton();
JButton button3 = new JButton();
```

3 JButton objects




```
public void start() {
    button1.setText("Start");
    button2.setText("OK");
    button2.setForeground(Color.yellow);
    button2.setBackground(Color.black);
    button3.setIcon(new ImageIcon("tfh-logo_new.gif"));

}
```

Type String

```
String text = „    Niceto see you!    ";
text=text.trim();                  // → "Nice to see you!"
text=text.toUpperCase()            // → "NICE TO SEE YOU!"
text=text.toLowerCase()            // → "nice to see you!"
c = text.charAt(3)                 // → 'e' (always start with 0!!)
text = "hello, " + text            // → "hello, nice to see you!
text=text.replace(',','-');        // → "hello - nice to see you!"
```

String Literals

- special feature of class String:
  String objects can be "directly denoted"

```
String greeting = "Hi, old folks!"
```

- the text in quotes is called a Literal

- abbrieviating

```
String greeting =
     new String("Hi, old folks!");
```

The Java Documentation

String (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://java.sun.com/j2se/1.4.2/docs/api/

http://java.sun.com/j2se/1.5.0/docs/api/

java.beans.beanc
java.io
java.lang
java.lang.ref
java.lang.reflect
java.math

Package
Process
Runtime
RuntimePermissi
SecurityManager
Short
StackTraceEleme
StrictMath
String
StringBuffer
System
Thread
ThreadGroup
ThreadLocal
Throwable
Void

Exceptions

Overview Package Class Use Tree Deprecated Index Help *Java™ Std.*

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Class String

```
java.lang.Object
  └java.lang.String
```

All Implemented Interfaces:

CharSequence, Comparable, Serializable

public final class **String**
extends Object
implements Serializable, Comparable, CharSequence

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

String (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://java.sun.com/j2se/1.4.2/docs/api/ Go

java.beans.bear
java.io
java.lang
java.lang.ref
java.lang.reflect
java.math

Package
Process
Runtime
RuntimePermiss
SecurityManage
Short
StackTraceElem
StrictMath
String
StringBuffer
System
Thread
ThreadGroup
ThreadLocal
Throwable
Void

Exceptions

Method Summary

char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contentEquals**(StringBuffer sb) Returns `true` if and only if this `String` represents the same sequence of characters as the specified `StringBuffer`.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.

cs101: Type Console

```
//...one of the cs101 library types:

    Console.print("Hel");
    Console.print("lo");
    Console.print(" World");
    Console.println("!");
    Console.println("Good morning!");

    Console.readln();
```

```
Hello World!
Good morning!
```

The cs101 Documentation

http://www.tfh-berlin.de/~ischmied
/Pr15/Uebungen/cs101Lib/Doc/

Console (Documentation for CS101 Package Problem Set) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address F:\Prog1\Pr15\Uebungen\cs101Lib\Doc\index.html Go

CS101 Package

All Classes

ClientDialog
ClientWire
Coerce
ColorField
ConnectionRejecte
Connector
Console
Console
CreationException
DefaultFrame
DefaultGameFrame

Constructor Summary

Console() Create and display a new Console.

Method Summary

void	**print**(java.lang.String s) Write a line to Console.
void	**println**(java.lang.String s) Write a String to Console.
java.lang.String	**readln**() Read a line from Console.

My Computer

Example Person

Imagine a type for storing and working on personal data as follows:

```
public class Person {
    String name;
    String firstname;
    String address;
    String telefon;
    String fax;
    String bankaccount;
}
```

```
Person eva = new  Person();
eva.name = "Mayer";
eva.firstname = "Eva Maria";
Person adam = new Person();
adam.name = "Mayer";
adam.firstname = "Sven Adam";
```

Object State: Attributes

- Attributes are variables defined on class level. They belong to the type's blueprint.

```
class Person {
  String name;
  //....
}
```

- The state of an object of type Person at a given time is defined by the values of its attributes :

```
eva. name = "Müller";    // Initialisation
eva. name = adam.name;   // Marriage with Adam
```

- Attributes define the state of an object.
- Methods may use and modify this state.

Initialising Objects

- There is an initial state for each object:
  - start value of ist attributes.
- Java implicitly initializes all attributes
  - with default values (0, false, ' ', null)
- Explicit Initialization:
  - combined with attribute declaration
  - by initializing attributes "from outside"
  - through constructors

```
public class Person {
   int age = 18;  // explicite Initialisation
   int children;  // implicitely: 0
}
```

```
Person anna = new Person();
anna.children = 10;  // Initialisation from outside
```

(

Constructors

- constructors are methods that are called, when an Object is created (new)
- A constructor has the classe's name and no return type
- A constructor may have parameters
- constructors can be overloaded.

```
public class Person {
   int age, children;
   String firstname, name;

   public Person() {    // Standard constructor
     age = 18;
     name = "Mustermann";
   }
   public Person(String secondname) {
                      // additional constructor
      age = 18;
      name = secondname;
   }
}
```

So, now we can create and initialize all the objects we would like to use in our "own universe"....

After the break: let's learn about interfaces.

Create Your Own Universe

- Classes and Behavior
- Types and Objects

Review Questions

- What is the difference between a function and a void method?
- What does the return statement do?
- What is the meaning of `private`?
- How do you determine the type of a statement?
- Can there be, in a single program :
  - two classes with the same name?
  - two methods or attributes with the same name?
  - Two methods with the same name within the same class?

Method Overloading

- Methods are identified by their signature.
- I.e. two methods with the same name can coexist, if their parameter lists differ (in number or type)
- This is called **method overloading.**

- Oberloading is forbidden, if method calls are indistinguishable to the compiler,
  - if their signatures only differ in the return type or access modifier
  - if they can be matched through implicit parameter type casting
  - if they only differ in their static modifier.

Test: Overloading

Which methods can coexist?

```
1. public int m(int a, String b);
2. public void m(int a);
3. public static void m(float a);
4. public int m(int a, boolean b);
5. private void m(boolean b);
6. private void m(long a);
```

Using Overloading

```
public class BusinessCards
{   // standard version
    public static void printCards
        (String name, String address,
         int number)
    { ... }
    // more options
    public static void printCards
        (String title, String name,
         String company,
         String addres,
         int layout, int number)
    { ... }
}
```

I. Classes and Behavior

Method Specification

- All you need to know to call a method correctly is found in the **method header**:

```
public static int meth(char c)
```

- It is also called **method signature** (or footprint)
- Syntactically, a signature is a complete **method specification**.
- It does not tell you anything about the semantics → add comments!

Class Specification

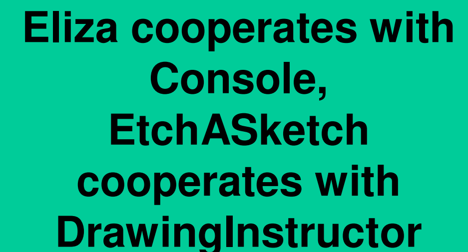
- The information needed to use a class properly is called its **export interface,**
- described by the **class specification.**
- It consist of all **public declarations**:
  - Attributes: type and name
  - Methods: signatures
  - Comments explaining the samantics

Classes "behave" and categorize

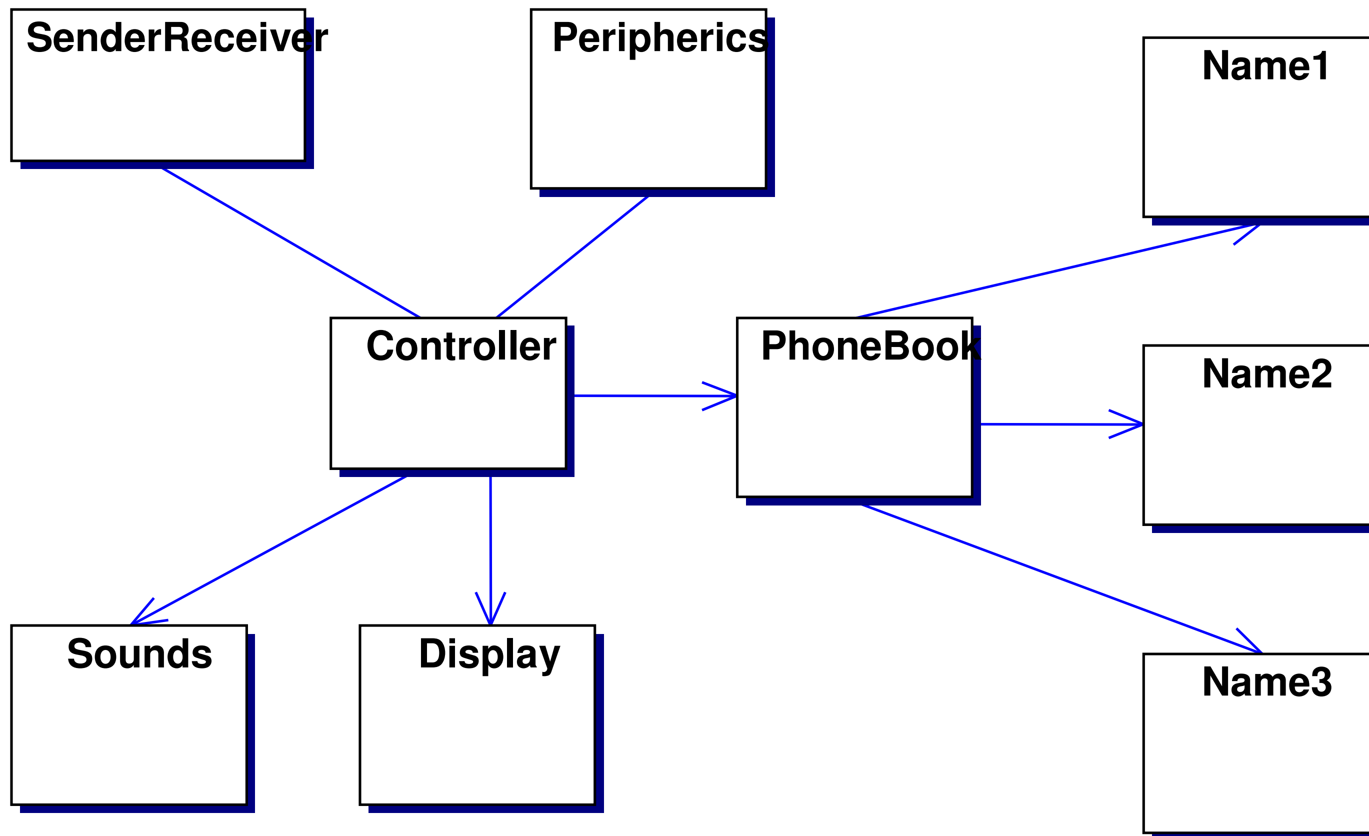
- To use a class means to call ist methods.
- Console "behaves":
  - if you call println, it will do output,
  - if you call readln, it will accept input and make it available to the caller.
- Console groups i/o behaviour
- Would you put a method called calculateMedian into Console?
  - No – it does not "fit".
  - Console methods deal with input and output.

Objects

- Objects in real life
  - contain information of a certain category
  - behave (work, react)
  - coorperate or interact
- Classes in a Program
  - store information
  - behave according to their methods
  - cooperate



Cell Phone:
System of Interacting Units



Controller (incoming call)

state: free / busy

rules of behaviour:

- incoming call notification (from Sender/Receiver)
  - → busy
  - - lookup name and personalized ringing tone
  - - display name
  - - sound ringing tone
- call rejected (from Peripherals)
  - - stop ringing
  - - display neutral
  - → free
- call accepted (from Peripherals)
  - - open connection
  - - display name and time lapse
  - - open mic and phones
- call finished (from Peripherals)
  - - close connection
  - - close mic and phones
  - - display neutral
  - → free

Display

state: current text, time lapse

rules of behaviour:

➢display call

- clear display
- display required text / screen

Phonebook

state: current entry

rules of behaviour:

➢lookup by number

- return name and ringing sound
- return null, if not found

➢lookup by name

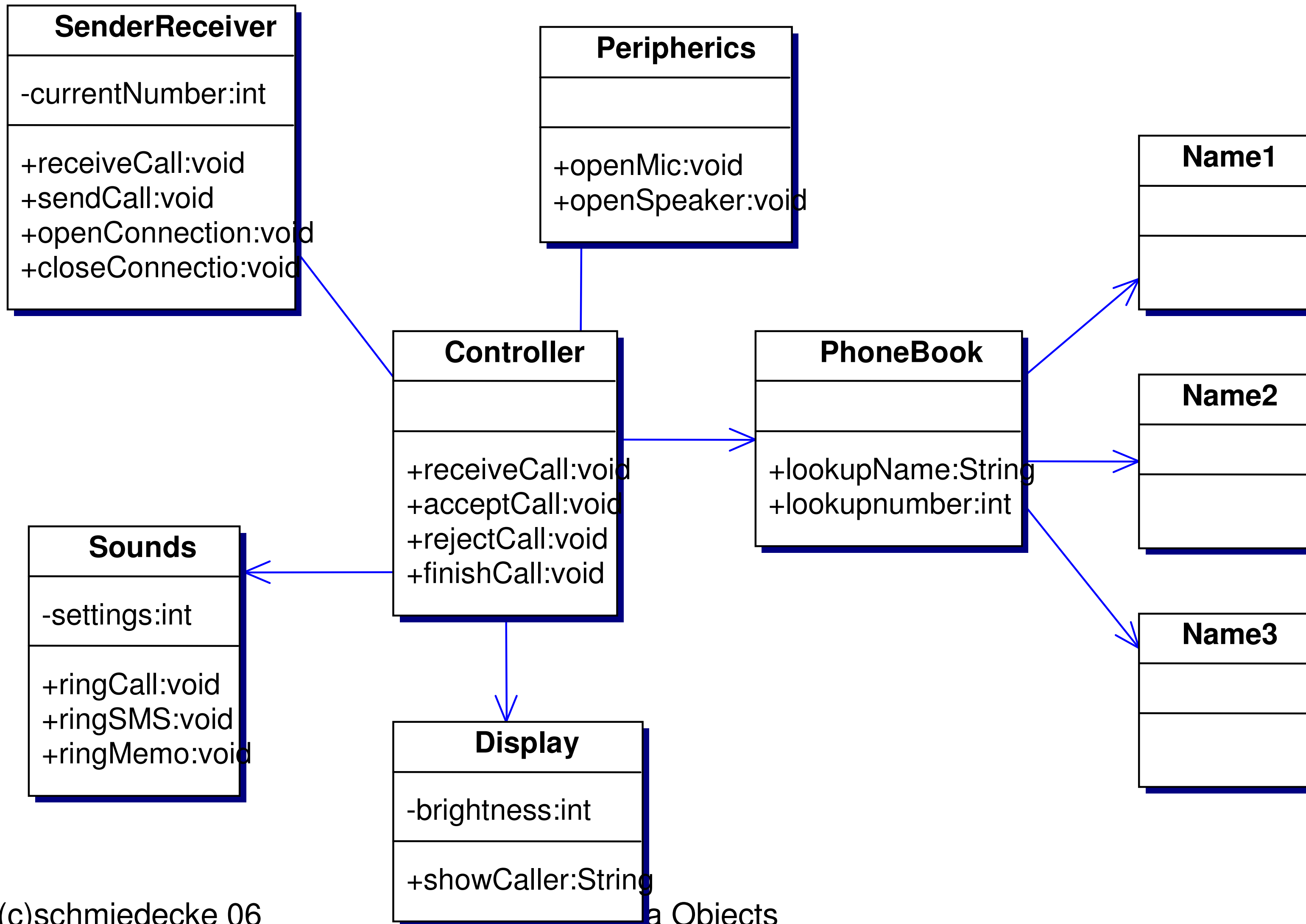
-return number
- return null, if not found

Sound

state: active / quiet, current sound

...

Cell Phone (Recieving call)



information

behavior

```
public class SenderReceiver {
  int currentCall = -100;
  public static void receiveCall(int number) {
    currentCall = number;
    Controller.receiveCall(number);
  }
  public static void sendCall(int number);
  public static void endCall();
  // ...
}
```

behavior

```
public class Controller {
  public static void receiveCall(int number) {
     Sounds.ringCall();
     String name = PhoneBook.lookupName(number);
     Display.showName(name);
  }
  public static void acceptCall();
  public static void rejectCall();
  public static void endCall();

  // ...
}
```

Programming interacting units

- I can program the Class Controller, if I have the specification of the classes Sound, Display, PhoneBook

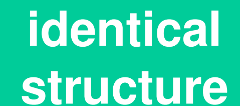
- In a System, each object contributes its behavior a the system, maybe using other objects' behavior.
- So a class can be considered an object...

II. Types and Objects

...Flaw in Concept

- Concept Class as Object credible → except for Name
- Just one class Name
- But we want many names in our phone book:

```
public class Name1 {
    static String name = "Helen";
    static int homeNumber = 007;
    static int cellNumber = 0815;
    editNumber();
}
public class Name2 {
    static String name = "Maurice";
    static int homeNumber = 2050;
    static int cellNumber = 0123;
    editNumber();
}
```



- *Problem1: program length ....*
- *Problem2: new entries*

Real Life Objects

- There are unique objects:
  - the Eiffel tower
  - the US president
  - the Control unit of your cell phone (within the system)
  - YOU...
- and there are many Objects that are instances of a type:
  - a lab workstation
  - a bicycle
  - a department
  - a phone book entry

Classes as Types

- Consider the unique object as a special case.
- Don't consider classes objects
- but as object blue prints,
  i.e. as **types:**

- A type defines
  - which information a "thing" can store → Attributes
  - and what can be done with this information
    → Methods

Class "Name" as Type

- A type defines
  - which information a "thing" can store → Attributes
  - and what can be done with this information → Methods
  - *another way of putting it: State Domain and Behavior Pattern*

Example: **Type Name**

State: name, home number, cell number, office number, group, ringing tone.

Behavior: edit name, edit number, add number, set group, set ringing tone, get home number, get cell number, ...

A Name Object (or "Name Instance")

An Object is an Entity which

- exists at runtime
- has a type
- possesses a *specific state* and
- shows a *specific behavior*

Example: **Name Object IlseSchmiedecke**

State: name = "ilse schmiedecke“
home number = 001-501-123456
cell number = 001 -251-665588
group = 24
ringing tone = emergency
Behavior: set group, get ringing tone…..

"Getter and Setter Methods"

- The state of an object is given by ist attribute values.
- Avoid making an attribute public.
- Define a set of functions to read the attributes - Getter:

```
public String getName();
public int getCellNumber();
```

- Define a set of void methods to change the attribute values - Setter:

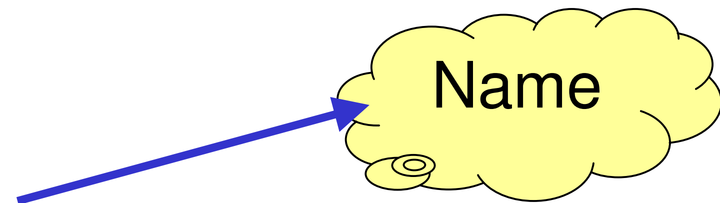
```
public void setName(String name);
public void setCellNumber(int number);
```

Creating Objects from Classes

- Type defined as a class:
-
```
public class Name { .... }
```

- Object Instantiation: operator new

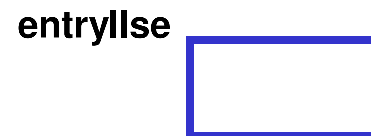
```
new Name()
```



  – creates a new object of type Name *somewhere in memory*
  – evaluating the new-expression yields a reference to the new object

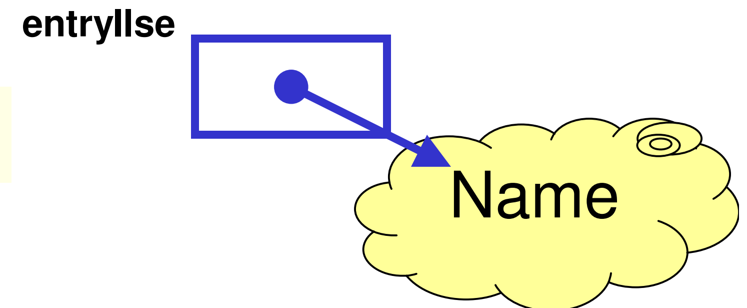
- Variable of Class Type

```
Name entryIlse;
```



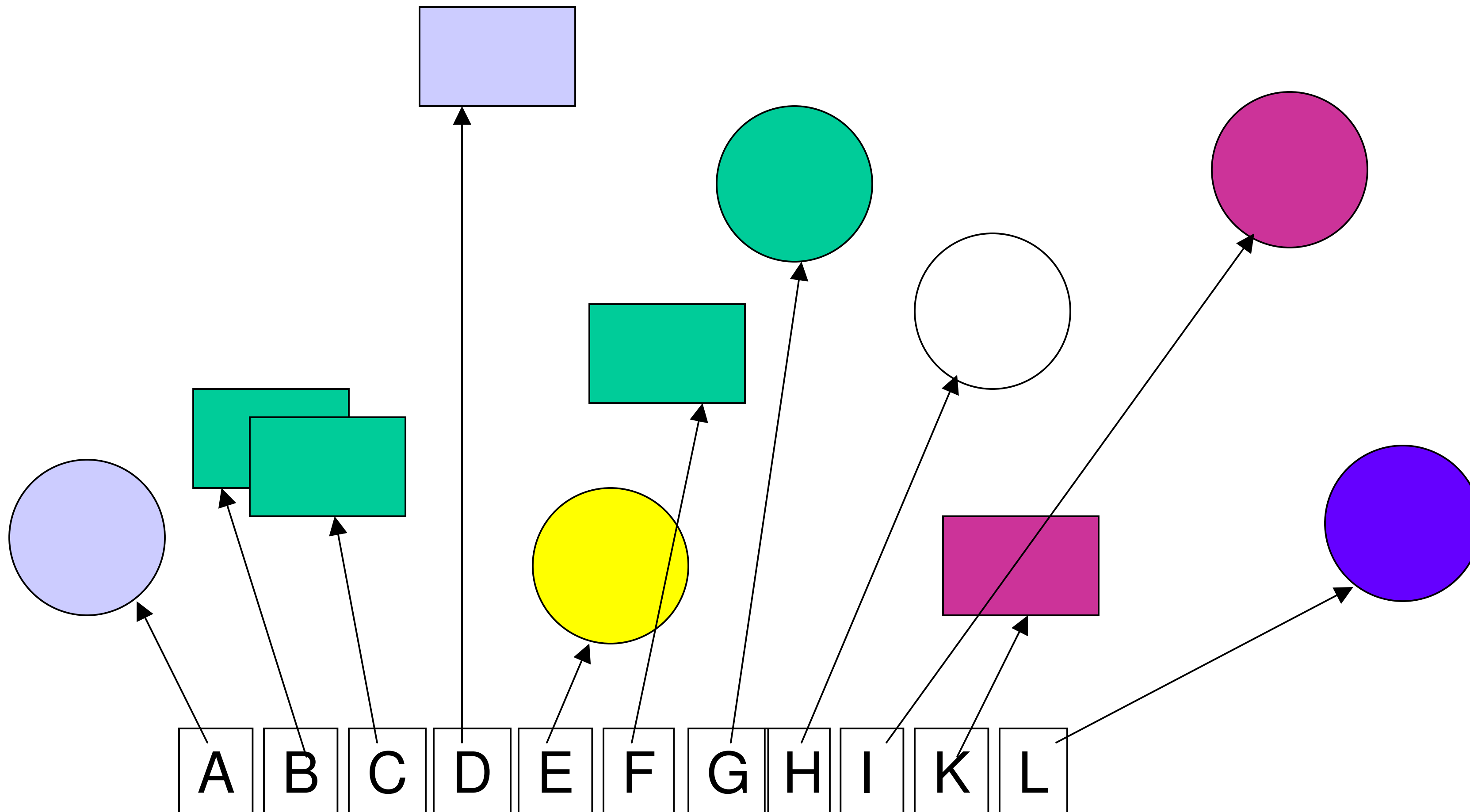
  - can store an object reference of given type:

```
entryIlse = new Name();
```



Objects and Variables

`new` can be used to create new Objects



Variables attach identifiers to them, so that they can be used.

Using different Objects

```
public class PhoneBook{
   public static void main(String[] args) {
      Name entryIlse = new Name();
      Name entryUwe = new Name();

      entryIlse.setRingingTone(emergency);
      entryUwe.setRingingtone(badinerie);
   }
}
```

not:
`Name.setRingingTone()`

- Variable (object reference) instead of class name
  action affects only one object of the class
- **`static`:**
  Attribute or method belongs to the class,
  is shared by all objects of the class,
  can even be used without any object (class name)

⇒ stop using **`static`** from now on!

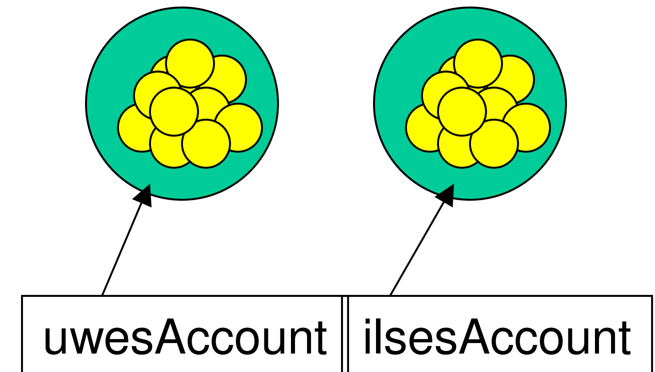
Static and Instance class Members

```
class Name {
  static int ownNumber;       // static, same for all Name objects
  static int defaultTone;
  static int defaultGroup;


  String lastName;            // object specific
  int homeNumber;             // or "instance specific"
  int officenumer;
  int cellNumber;
  String email;

  public static void setOwnNumber(int number);
  public static void setDefaultGroup(int group);
  public static void setDefaultTone(int tone);

  public int getHomeNumber();
  public void setHomeNumber();
  public int getOfficeNumber();
  public void setOfficeNumber();
  //...
}
```

Using Object Variables

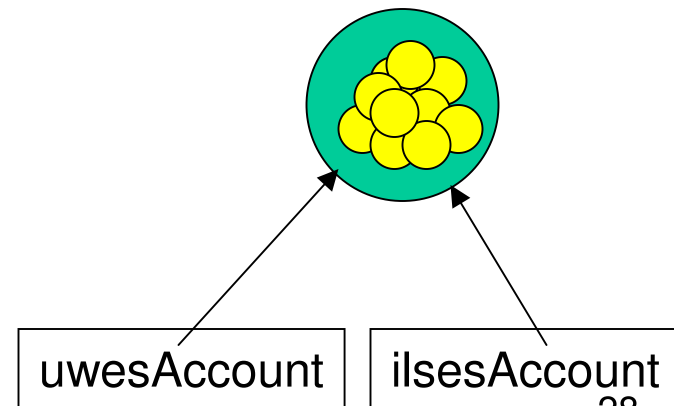
- **If two variables refer to different objects, their actions do not interfere:**

```
Account uwesAccount = new Account();
Account ilsesAccount = new Account();

ilsesAccount.withdraw(1000.00);
uwesAccount.withdraw(1000.00);
```



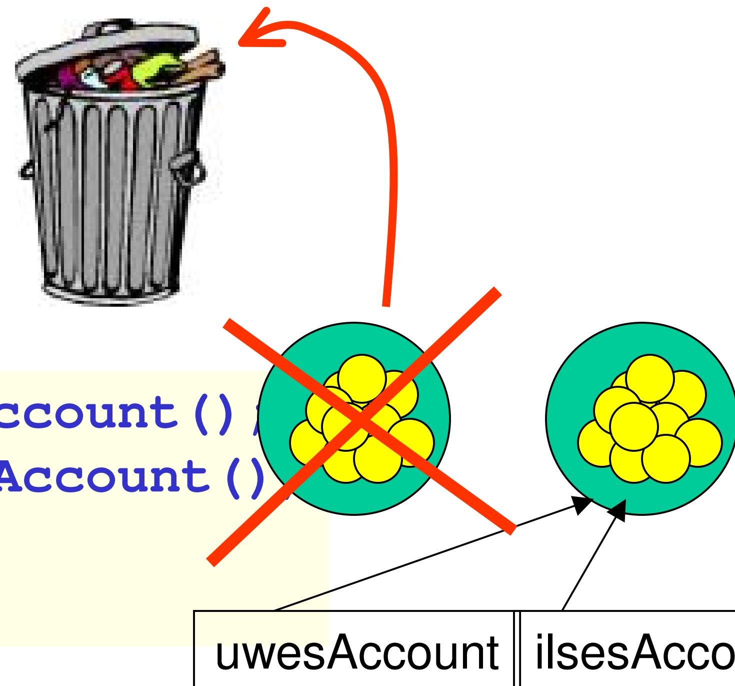
- **You can assign a variables to a variable, then they refer both to the same object:**

```
uwesAccount = ilsesAccount;

ilsesAccount.withhdraw(1000.00);
uwesAccount.withdraw(1000.00);
```



Loss of Objects

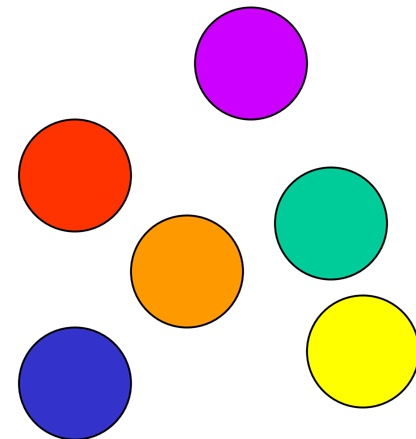
- Generally, objects live from their creation to the end of the program.
- If there is no reference to an object, it is lost (unreachable)
- In Java, a Garbage Collector collects these lost objects (at some time) and restores the storage space.




```
Account uwesAccount = new Account();
Account ilsesAccount = new Account();
uwesAccount = ilsesAccount;
// the old object is lost!
```

Juggling with Objects

```
Ball annaLeft, annaRight,
     timLeft, timRight;
annaLeft = new Ball("red");
annaRight = new Ball("blue");
timLeft = annaRight;
annaLeft = new Ball("yellow");
timRight = new Ball("green");
annaRight = annaLeft;
timLeft = new Ball("orange");
annaLeft = new Ball("purple");
```



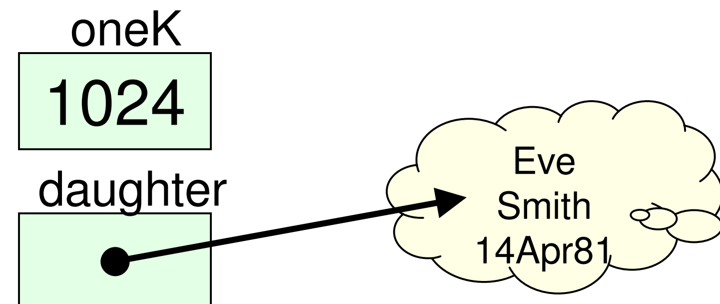
Who holds which ball? Which ones are lost?

Summary: Types

- Object types define object patterns.
- Variables of object types store references to objects.
- Using an object reference, you can call the object's methods.
- An object can have different states depending on ist attribute values.

- Primitive types have a direct representation on the machine– e.g. numbers.
- Values of primitve types are elements of a fixed set of values.
  They have no states.
- Variables of primitive types store values directly, not references to values.
- They cannot be used for method calls (but used with operators instead).

```
int oneK = 1024;
oneK.double() // error!

Person daughter = eve;
```



Some Java Standard Types

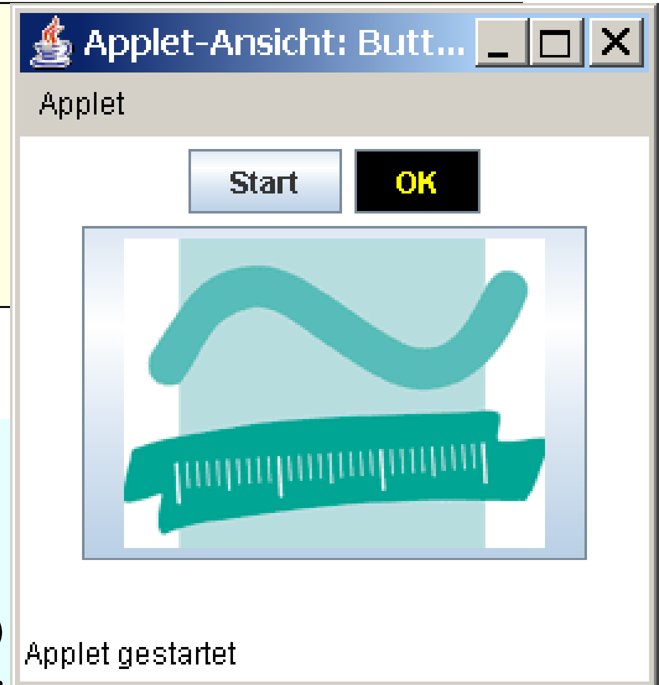
- `String`:
  - capitalize, get length, concatenate ...
- `DialogBox`
  - define text, show box, ...
- `JButton`
  - define text, change color, add action
- `Vector`
  - add object, find object, ...

....

Type JButton

```
JButton button1 = new JButton();
JButton button2 = new JButton();
JButton button3 = new JButton();
```

3 JButton objects




```
public void start() {
    button1.setText("Start");
    button2.setText("OK");
    button2.setForeground(Color.yellow);
    button2.setBackground(Color.black);
    button3.setIcon(new ImageIcon("tfh-logo_new.gif"));

}
```

Type String

```
String text = „    Niceto see you!    ";
text=text.trim();              // → "Nice to see you!"
text=text.toUpperCase()        // → "NICE TO SEE YOU!"
text=text.toLowerCase()        // → "nice to see you!"
c = text.charAt(3)             // → 'e' (always start with 0!!)
text = "hello, " + text        // → "hello, nice to see you!
text=text.replace(',','-');    // → "hello - nice to see you!"
```

String Literals

- special feature of class String:
  String objects can be "directly denoted"

```
String greeting = "Hi, old folks!"
```

- the text in quotes is called a Literal

- abbrieviating

```
String greeting =
     new String("Hi, old folks!");
```

The Java Documentation

http://java.sun.com/j2se/1.5.0/docs/api/

String (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://java.sun.com/j2se/1.4.2/docs/api/

java.beans.beanc
java.io
java.lang
java.lang.ref
java.lang.reflect
java.math

Package
Process
Runtime
RuntimePermissi
SecurityManager
Short
StackTraceEleme
StrictMath
String
StringBuffer
System
Thread
ThreadGroup
ThreadLocal
Throwable
Void

Exceptions

Overview Package **Class** Use Tree Deprecated Index Help *Java™*
Std.

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Class String

```
java.lang.Object
  └java.lang.String
```

All Implemented Interfaces:
CharSequence, Comparable, Serializable

public final class **String**
extends Object
implements Serializable, Comparable, CharSequence

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

String (Java 2 Platform SE v1.4.2) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://java.sun.com/j2se/1.4.2/docs/api/ Go

java.beans.bear
java.io
java.lang
java.lang.ref
java.lang.reflect
java.math

Package
Process
Runtime
RuntimePermiss
SecurityManage
Short
StackTraceElem
StrictMath
String
StringBuffer
System
Thread
ThreadGroup
ThreadLocal
Throwable
Void

Exceptions

Method Summary

char	**charAt**(int index) Returns the character at the specified index.
int	**compareTo**(Object o) Compares this String to another Object.
int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contentEquals**(StringBuffer sb) Returns `true` if and only if this `String` represents the same sequence of characters as the specified `StringBuffer`.
static String	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.

cs101: Type Console

```
//...one of the cs101 library types:

   Console.print("Hel");
   Console.print("lo");
   Console.print(" World");
   Console.println("!");
   Console.println("Good morning!");

   Console.readln();
```

```
Hello World!
Good morning!
```

The cs101 Documentation

http://www.tfh-berlin.de/~ischmied /Pr15/Uebungen/cs101Lib/Doc/

Console (Documentation for CS101 Package Problem Set) - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address F:\Prog1\Pr15\Uebungen\cs101Lib\Doc\index.html Go

CS101 Package

All Classes

ClientDialog
ClientWire
Coerce
ColorField
ConnectionRejecte
Connector
Console
Console
CreationException
DefaultFrame
DefaultGameFrame

Constructor Summary

`Console()`
Create and display a new Console.

Method Summary

void	`print(java.lang.String s)` Write a line to Console.
void	`println(java.lang.String s)` Write a String to Console.
java.lang.String	`readln()` Read a line from Console.

My Computer

Example Person

Imagine a type for storing and working on personal data as follows:

```
public class Person {
    String name;
    String firstname;
    String address;
    String telefon;
    String fax;
    String bankaccount;
}
```

```
Person eva = new  Person();
eva.name = "Mayer";
eva.firstname = "Eva Maria";
Person adam = new Person();
adam.name = "Mayer";
adam.firstname = "Sven Adam";
```

Object State: Attributes

- Attributes are variables defined on class level. They belong to the type's blueprint.

```
class Person {
  String name;
  //....
}
```

- The state of an object of type Person at a given time is defined by the values of its attributes :

```
eva. name = "Müller";    // Initialisation
eva. name = adam.name;   // Marriage with Adam
```

- Attributes define the state of an object.
- Methods may use and modify this state.

Initialising Objects

- There is an initial state for each object:
  - start value of ist attributes.
- Java implicitly initializes all attributes
  - with default values (0, false, ' ', null)
- Explicit Initialization:
  - combined with attribute declaration
  - by initializing attributes "from outside"
  - through constructors

```
public class Person {
   int age = 18;  // explicite Initialisation
   int children;  // implicitely: 0
}
```

```
Person anna = new Person();
anna.children = 10;  // Initialisation from outside
```

Constructors

- constructors are methods that are called, when an Object is created (new)
- A constructor has the classe's name and no return type
- A constructor may have parameters
- constructors can be overloaded.

```
public class Person {
   int age, children;
   String firstname, name;

   public Person() {   // Standard constructor
     age = 18;
     name = "Mustermann";
   }
   public Person(String secondname) {
                       // additional constructor
      age = 18;
      name = secondname;
   }
}
```

So, now we can create and initialize all the objects we would like to use in our "own universe"....

After the break:
let's learn about interfaces.