Tutorial

Object-Oriented Concepts for Distributed Systems I (OODS I)

Part I: Java

1 Organization

A.1 Part I: Java

- Java language, Packages, Applets and Applications
- Inheritance, Interfaces, Inner Classes, Abstract Classes, Final Classes and Methods, Exceptions
- AWT, Threads

A Organization

A.1 Part II: Distributed Programming

- Streams, Sockets
- Serialization, Classloader
- RMI
- ORB

A Organization

A.1 Part III: CORBA

- ...

A Organization

A.1 Part IV: Jini

- ...

A Organization

- You don’t have to hand in programming assignments that are labeled with "0 points" (e.g., assignment 1).
- We will reserve part of the CIP pool for OODS.
- A tutor will answer your questions and help you with the practical programming work in the CIP pool.

2 Overview of Todays Tutorial

- Introduction to the Java System and Language
  - Java vs. C
  - Classes and Objects
  - Methods and Constructors
  - Packages
  - Encapsulation
- Applets and Applications

3 What's Java all about?

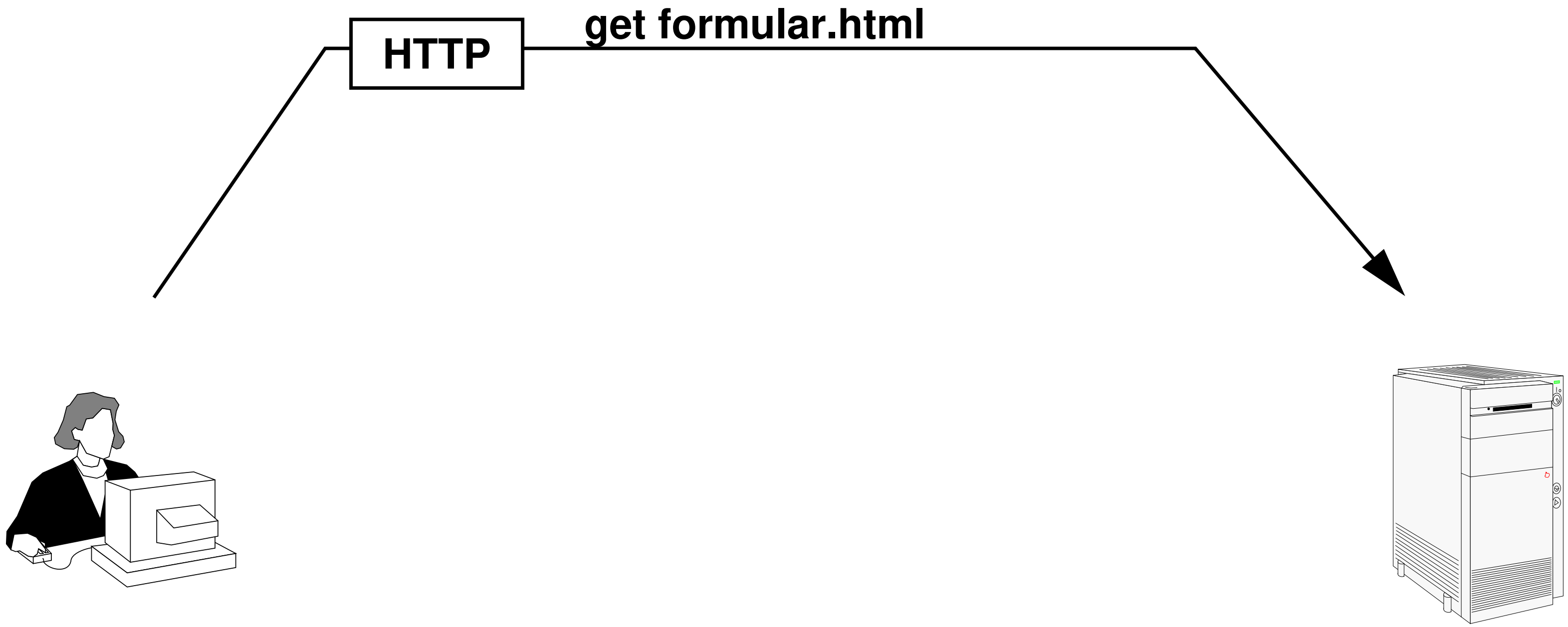
◆ At first sight:

- a language to create cool animations for web pages

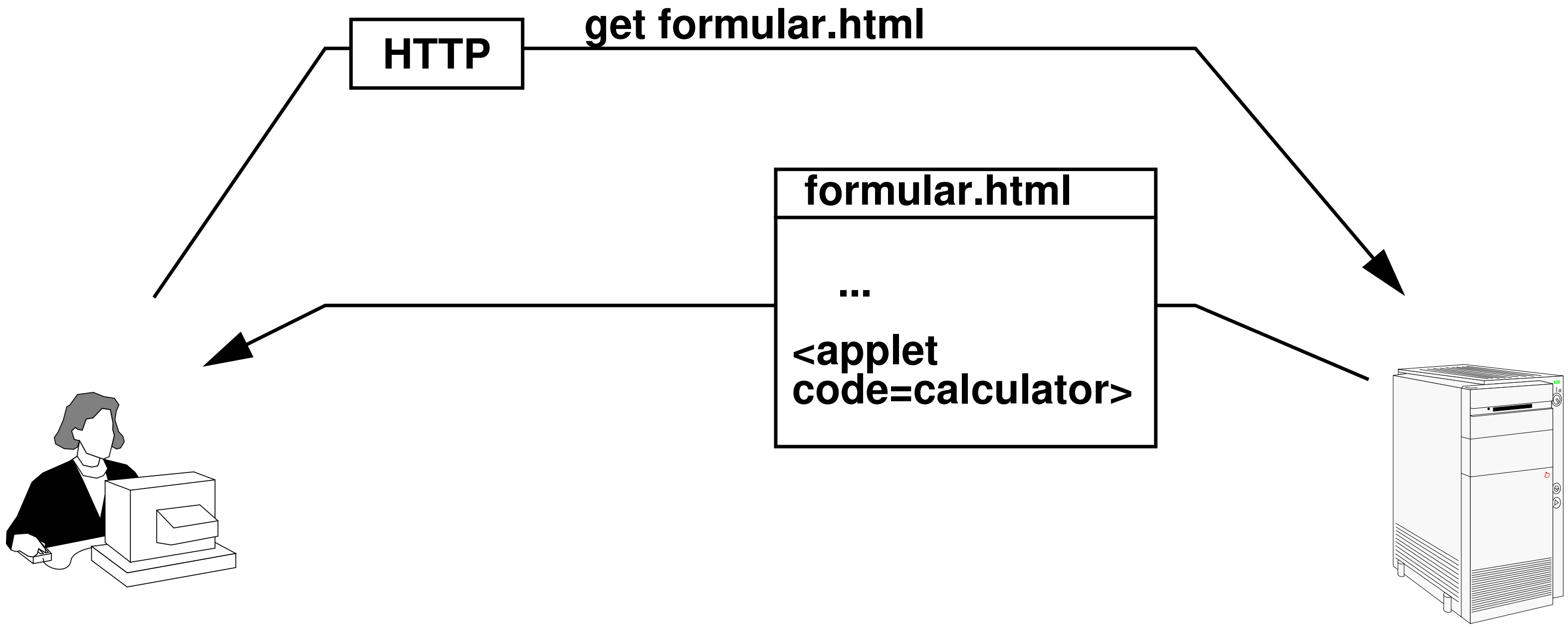
3.1 ... but more importantly

- an object-oriented language which is
  - simple
  - portable
  - robust
- a secure run-time system
  - execute arbitrary programs without any risk of damages
  - control access to local resources (e.g. files) and to the network

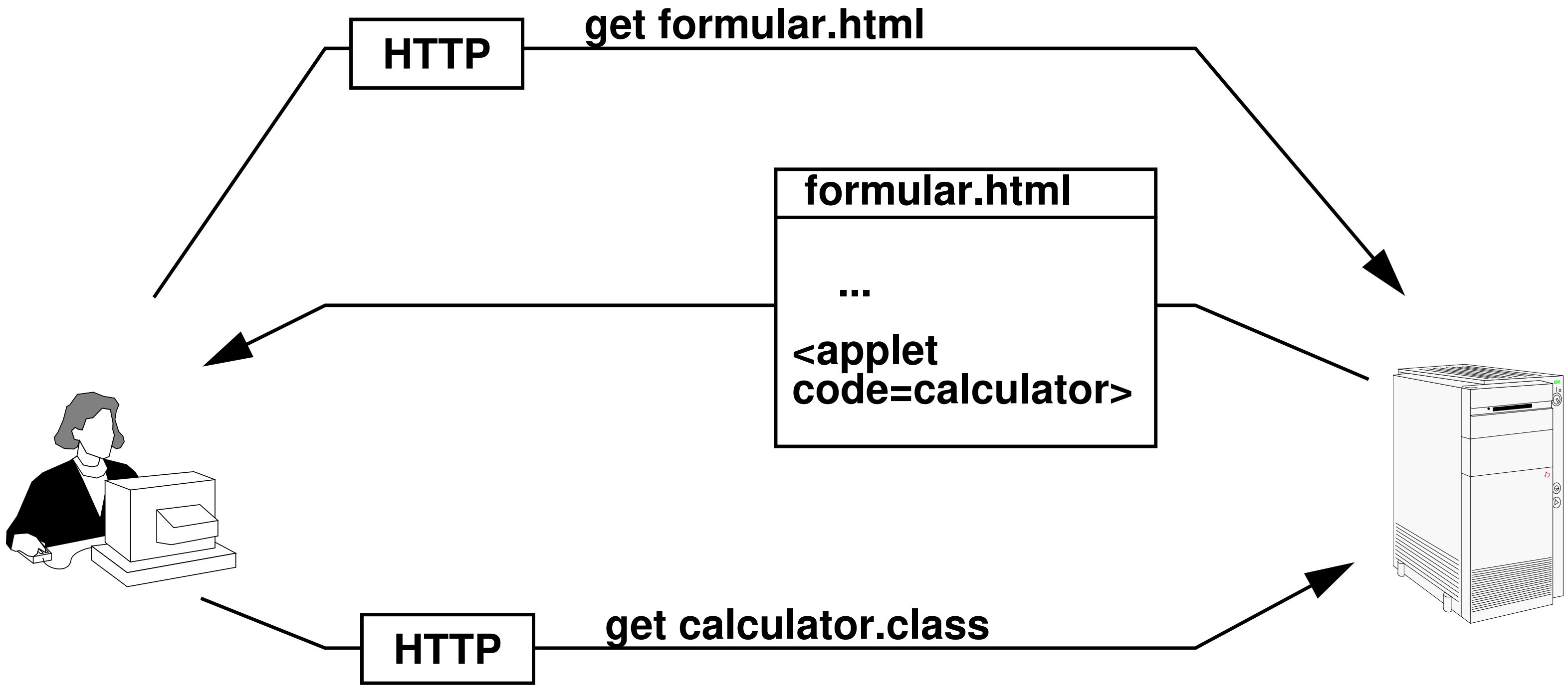
3.2 Java and the World Wide Web



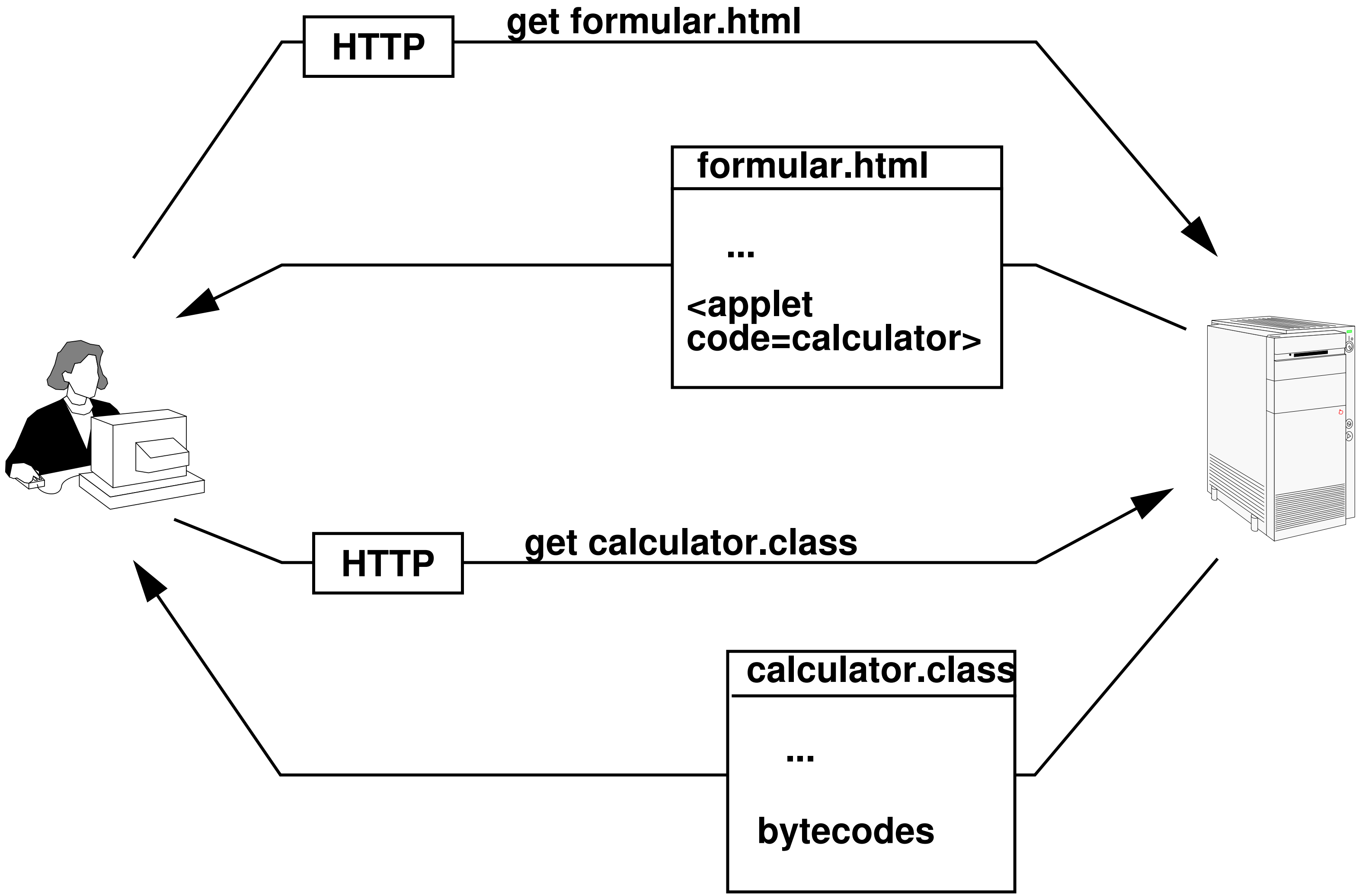
3.2 Java and the World Wide Web



3.2 Java and the World Wide Web

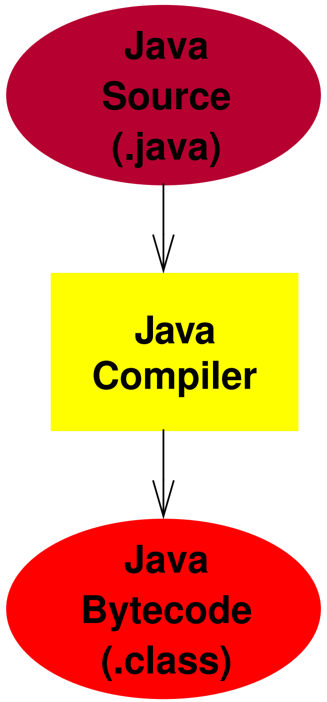


3.2 Java and the World Wide Web

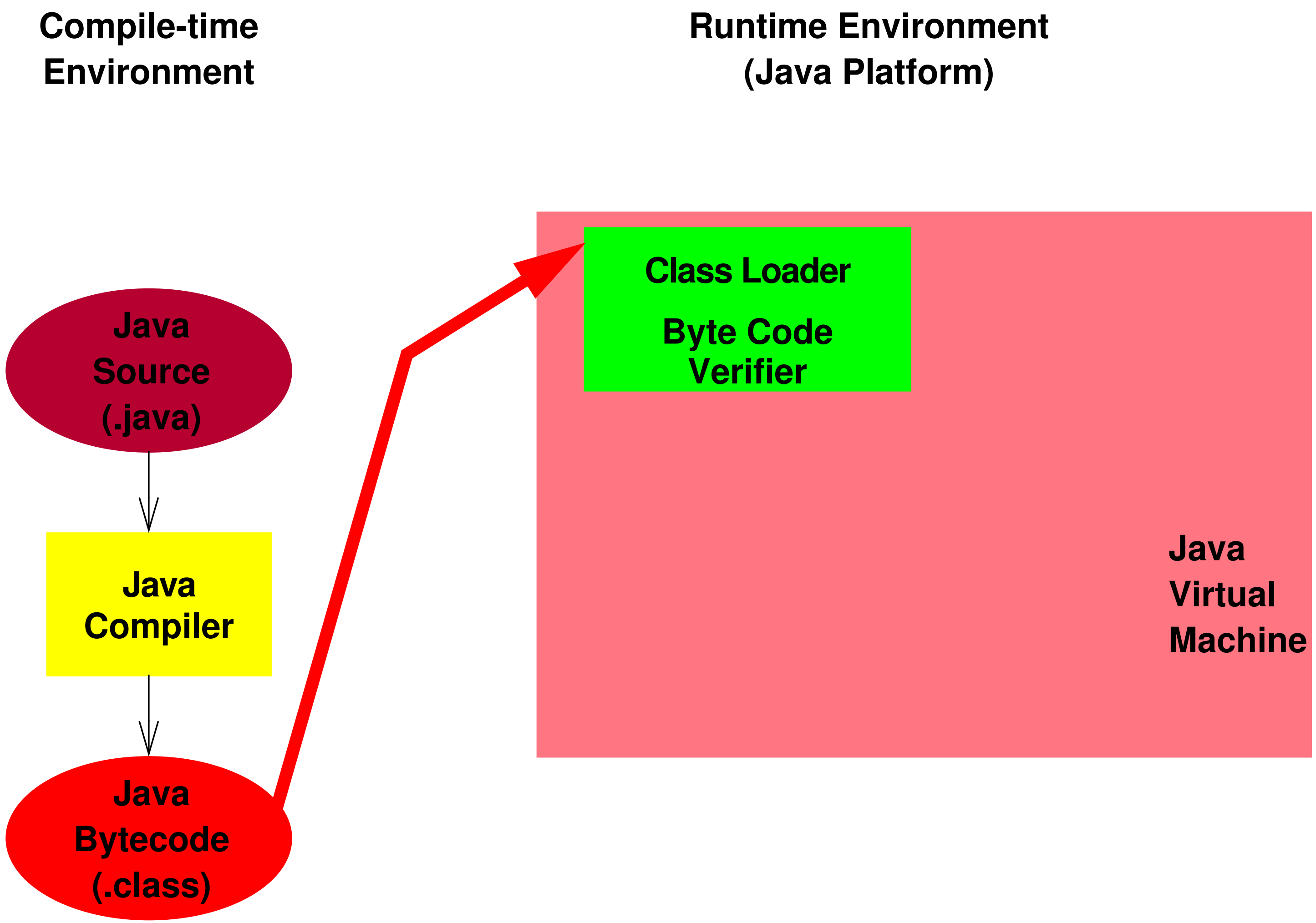


3.3 The Java System

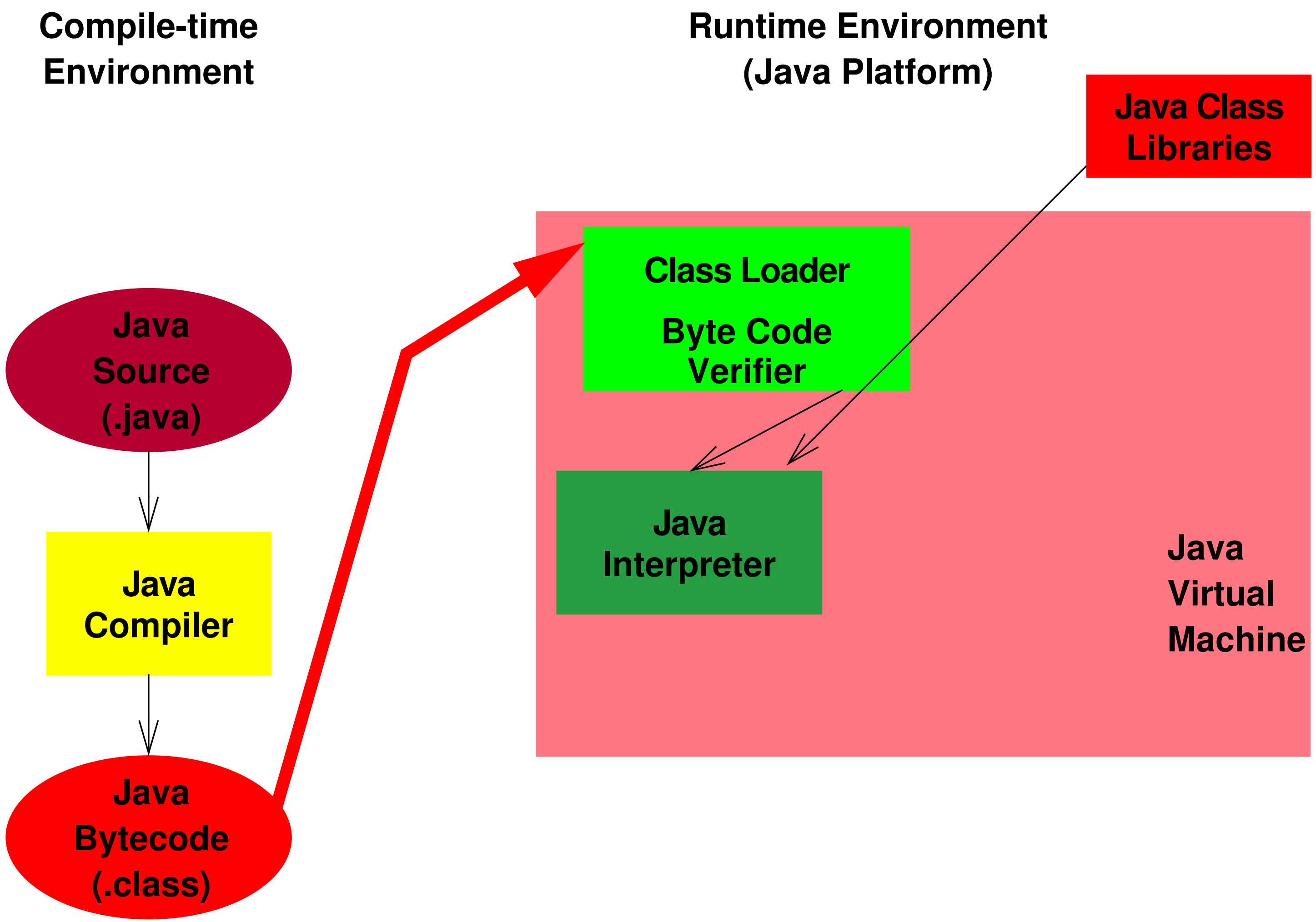
Compile-time Environment



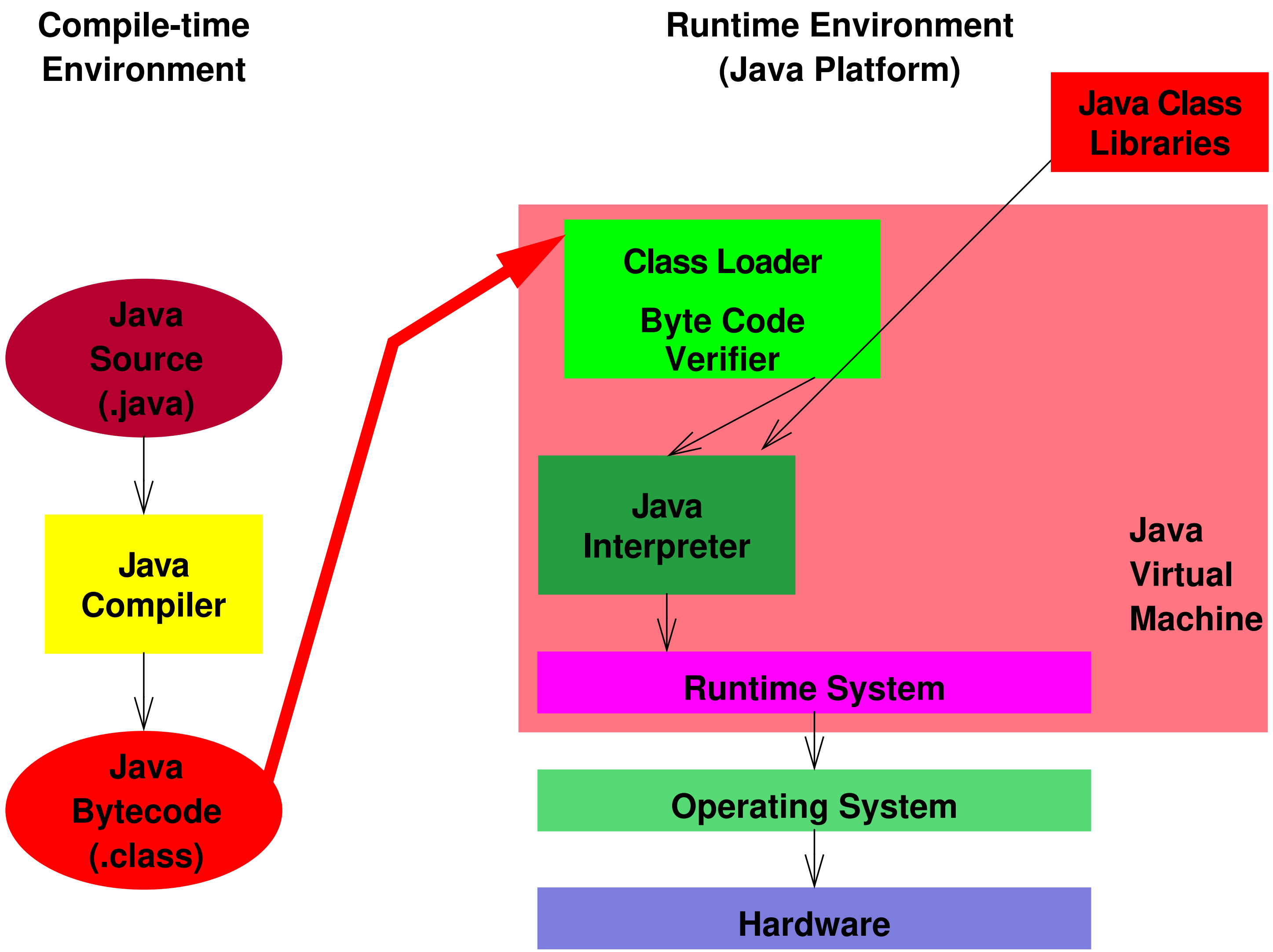
The Java System



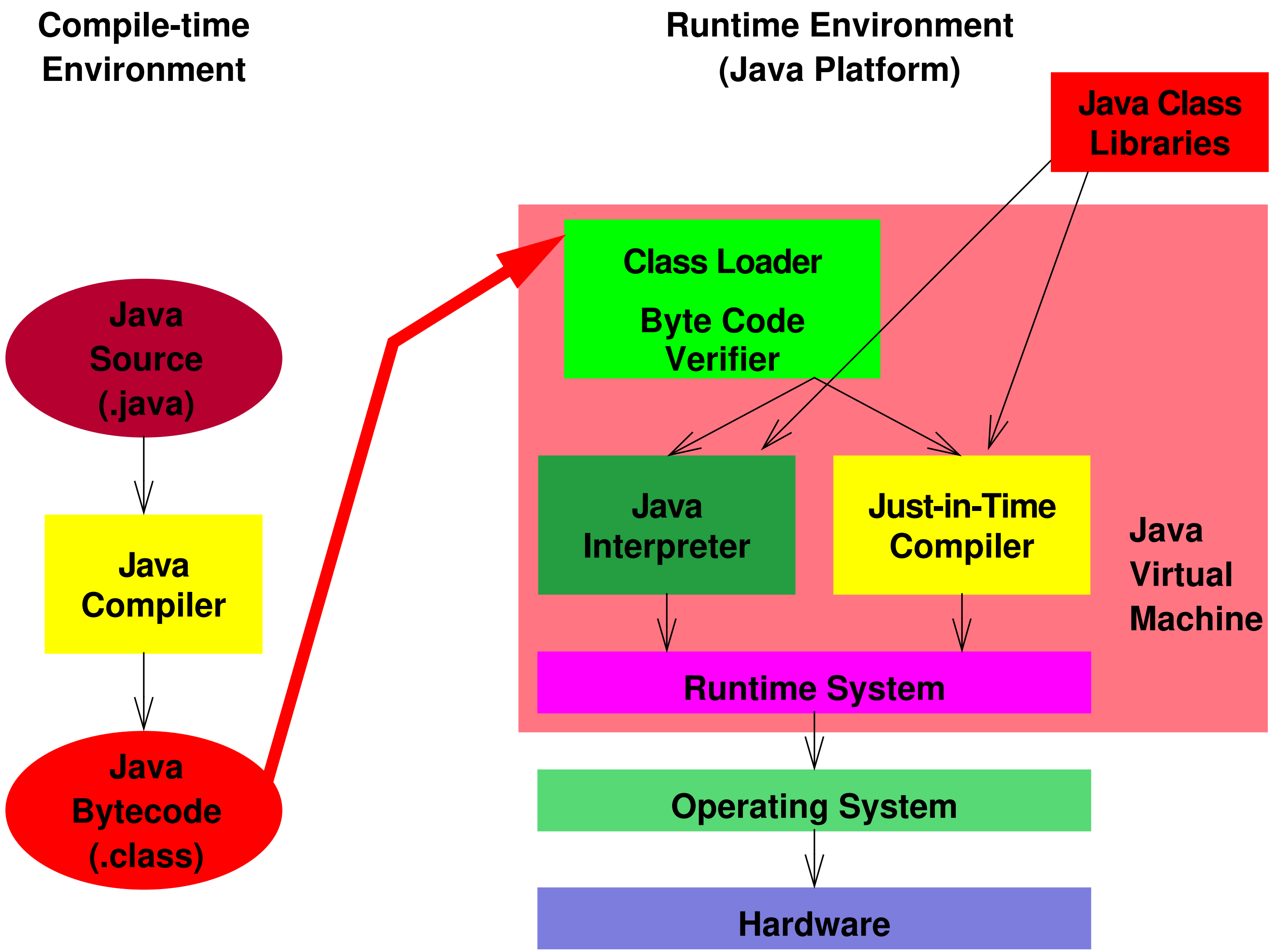
3.3 The Java System



The Java System



3.3 The Java System



4

Java vs. C/C++

- no pre-processor (#define)
- no typedef, union, enum, struct
- no pointer arithmetic
- no functions or procedures
- no global variables

4.1 Data types

- Primitive data types:
  - byte, short, int, long (8/16/32/64 bit)
  - float, double (32/64 bit)
  - char (Unicode, 16 bit)
  - boolean (true, false)
  - "ordinary" operators: +,-,+=,<<,...

- Differences to C:
  - no "unsigned"
  - special "boolean" type and Unicode-char type
  - no **`while (x--)`**, use **`while (x-- != 0)`**
  - no **`if (ptr)`**, use **`if (ptr != null)`**

4.2 Comments

- one line comment

```
// this is a comment
```

- multi-line comment

```
/* This comment
 * uses multiple
 * lines.
 */
```

- documentation comment (generate docs with the **`javadoc`** tool)

```
/** This comment
 *  contains program
 *  documentation.
 */
```

4.3 Control structures

- Control structures as in C:
  - for (... ; ... ; ...) { ... }
  - while (...) { ... }   or   do { ... } while (...);
  - switch (...) { case .... }
  - if ....
  - return, break, continue

- Differences to C:
  - no goto
  - labelled break und continue:

```
outerloop: for ( .... ) { // this is "outerloop"
    while ( .... ) {
        if ( ... ) break outerloop;// leave outerloop
    }
}
```

4.4 Strings

- Good support for string handling
- Data type **`String`**

```
String hello = "Hello";
```

- Concatenation with "+"

```
String world = "World";
String greeting = hello + " " + world + "!";
System.out.println(greeting);
```

Output:

Hello World!

4.4 Strings (2)

- Concatanation works also with int ,float,...

```
String test = "Result: " + (2 + 3);
```

Contents of test: "**`Result: 5`**"

- **Important: Strings are constant.**

5 Arrays

- Declaration with ***`type`* `name[]`** or ***`type`* `[] name`**
- Allocation with **`new`**, no free neccessary (garbage collector)

```
int numbers[] = new int[100];
```

- Element access as in in C: **`name[index]`**

```
for (int n=0; n<100; n++) numbers[n] = n;
```

- Atomatic range checking

5.1 Arrays (2)

- Declaration und initialization

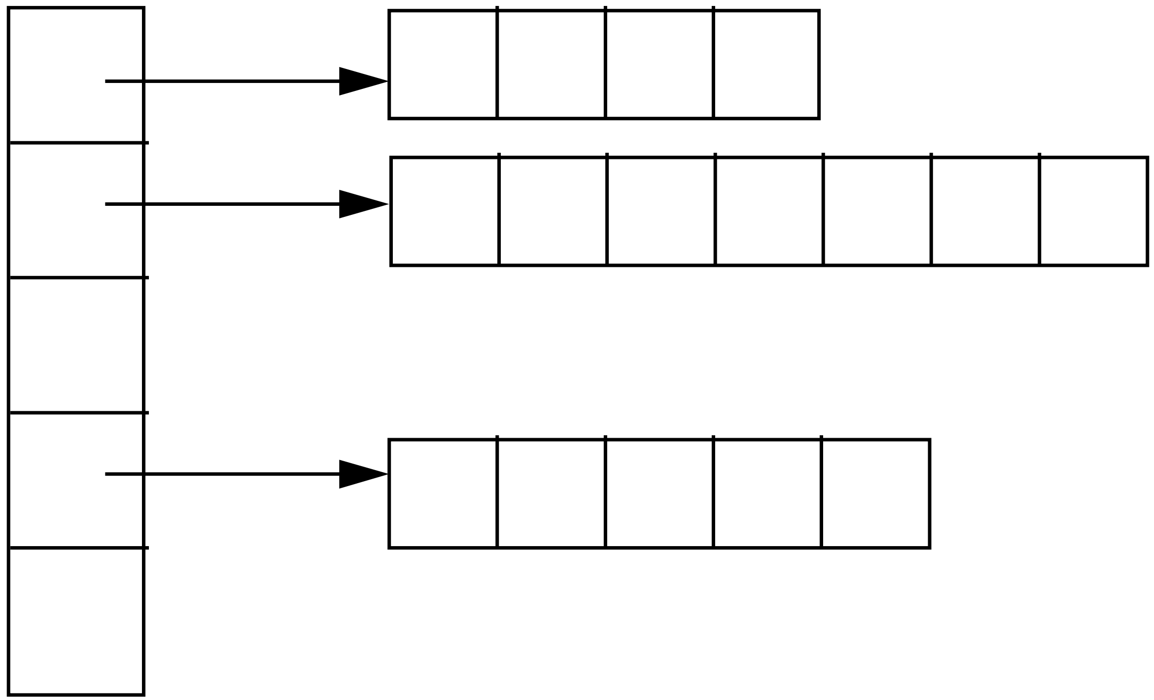
```
int [] firstPrimes = { 2, 3, 5, 7 };
```

- Access to array length with **`length`**

```
for(int i=0; i<firstPrimes.length; i++) {
    System.out.println(firstPrimes[i]);
}
```

5.2 Multi-Dimensional Arrays

- Array elements can be arrays



```
int [][] matrix = new int[5][4];
```

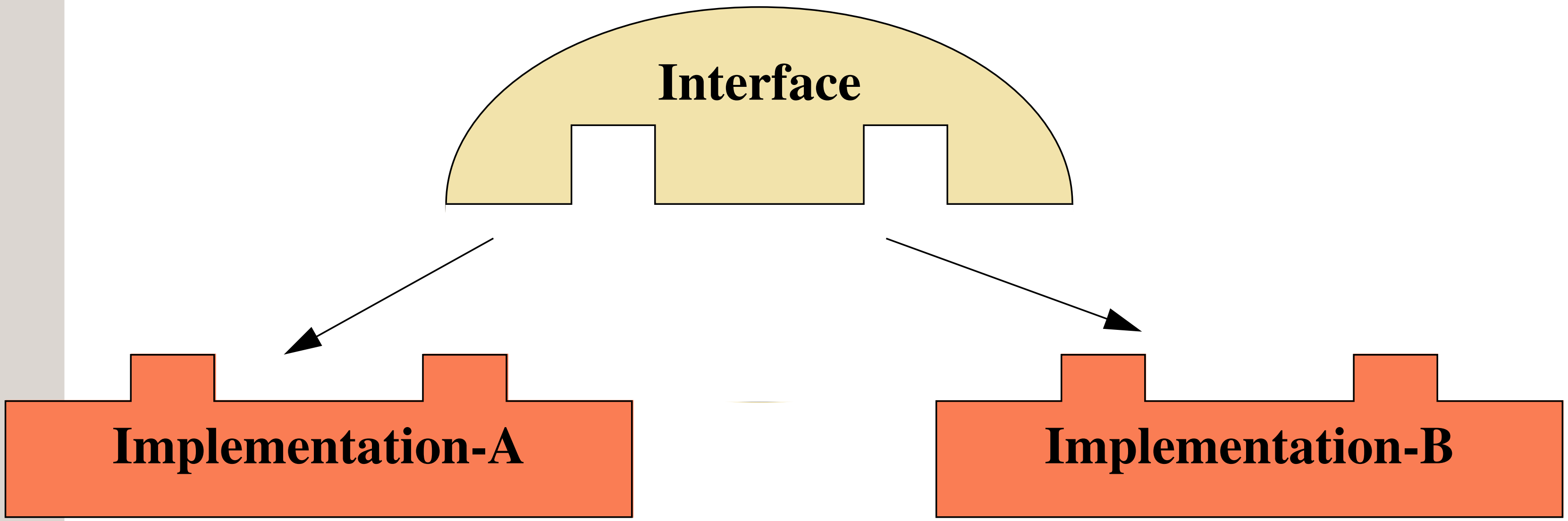
- fundamental concepts of object-oriented programming:

abstraction

encapsulation

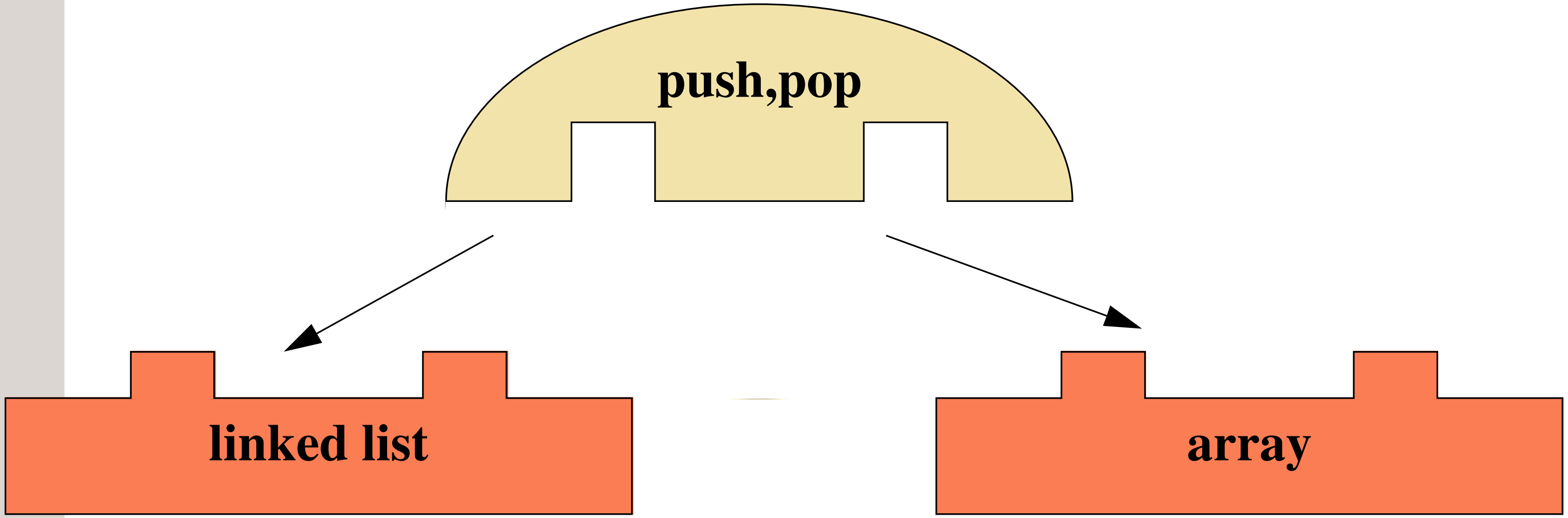
6.1 Abstraction

- Separation
  - interface (what can be done) and
  - implementation (how is it done)

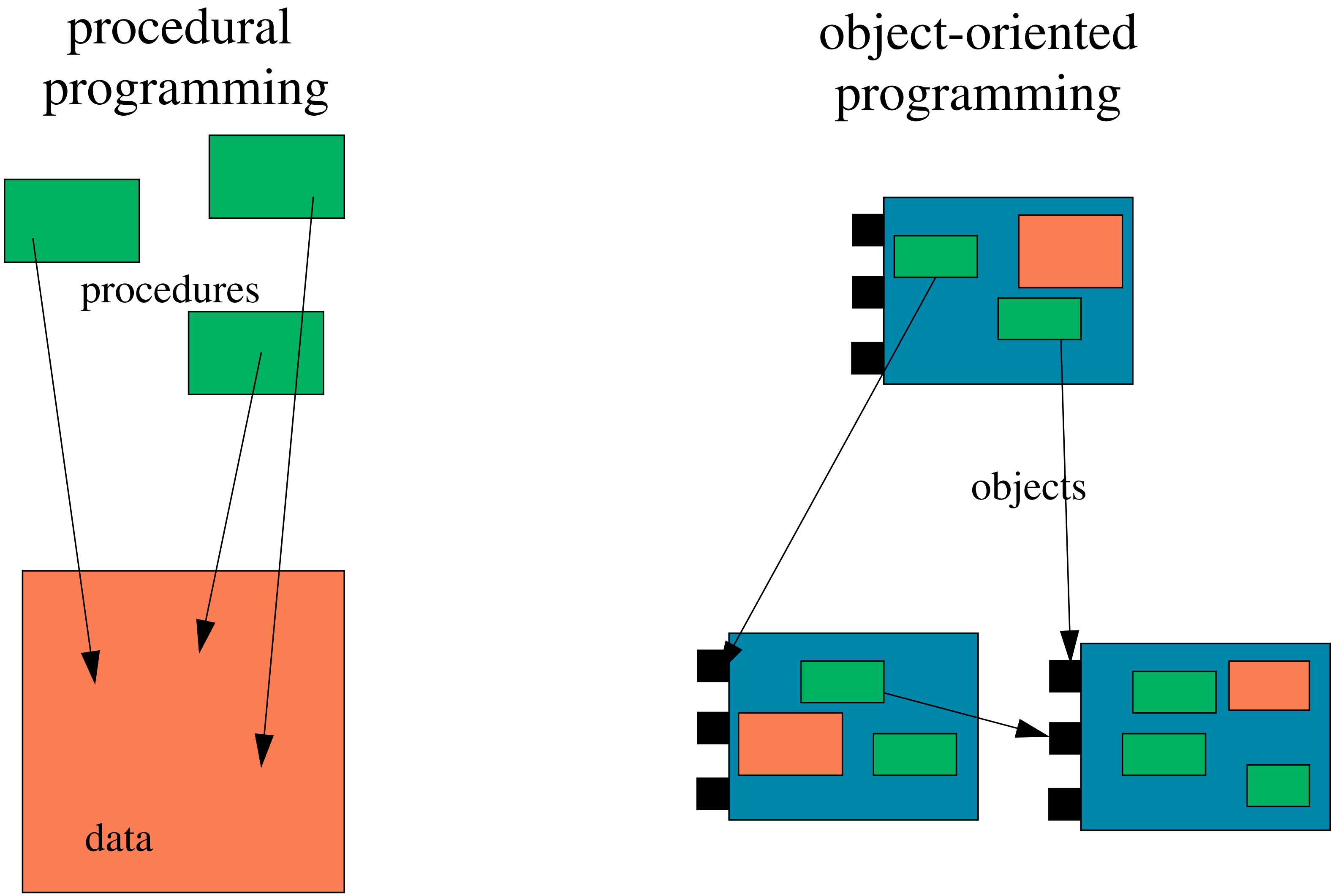


6.1 Abstraction

- Example: stack
  - interface: push, pop
  - implementation: linked list, array

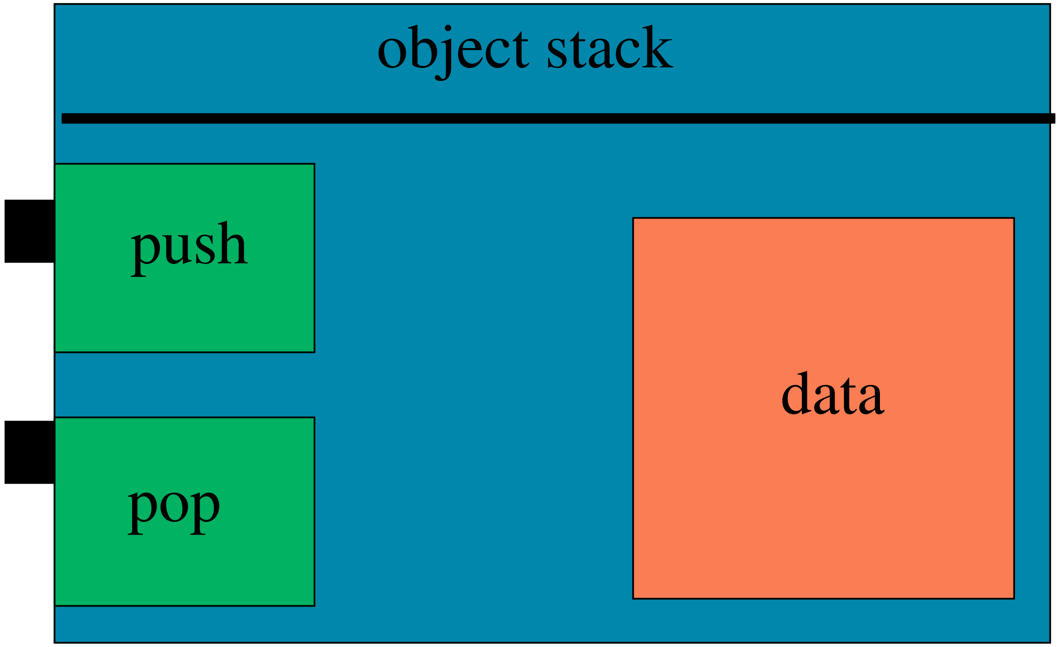


Encapsulation



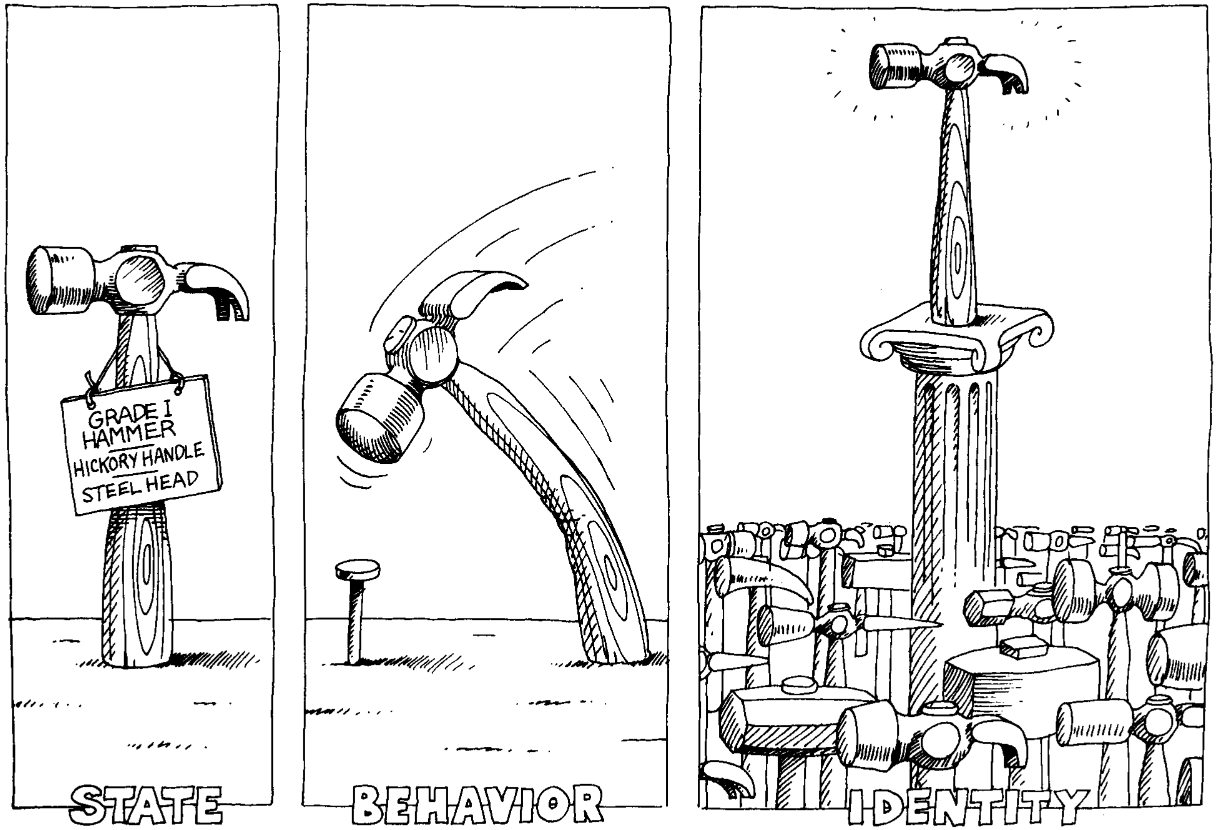
6.2 Encapsulation

- Objects: encapsulated data structure, consisting of
  - Data (instance variables, attributes)
  - Methods (operations)



- Encapsulation helps building abstractions

6.3 Properties of Objects



6.4 The Class

- objects are *instances* of a class
- the class determines internal structure and interface of the objects

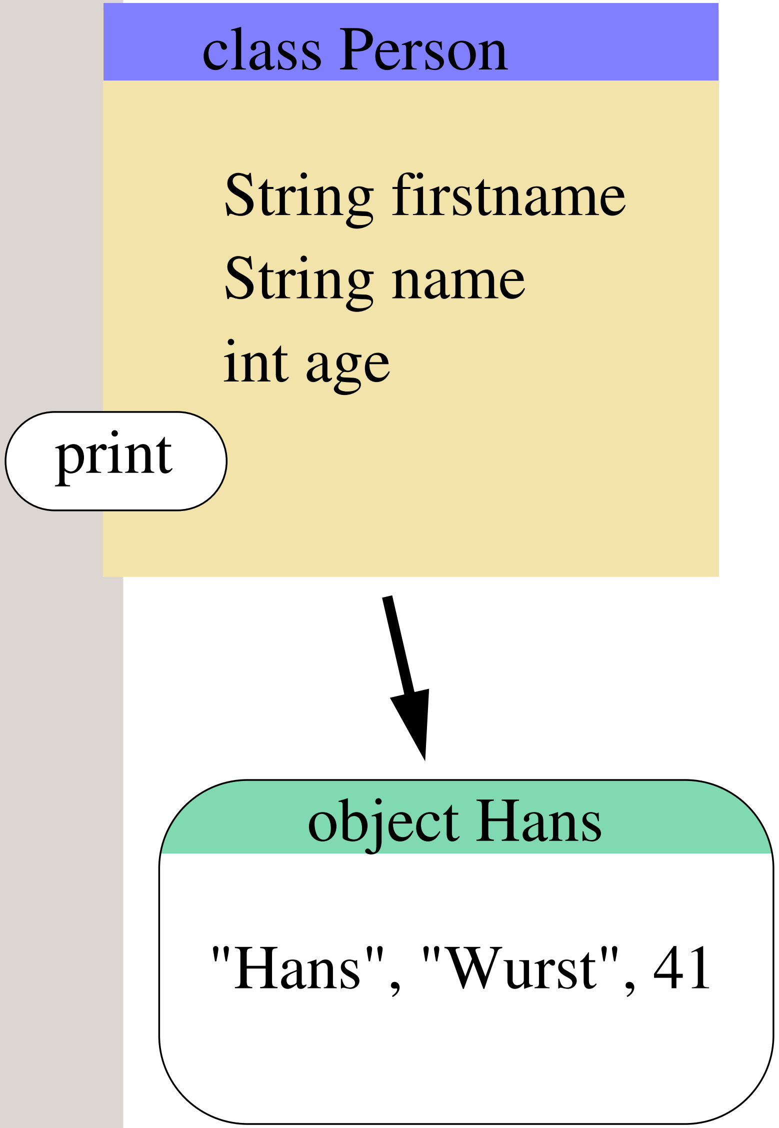
6.5 Class - Example

class Person

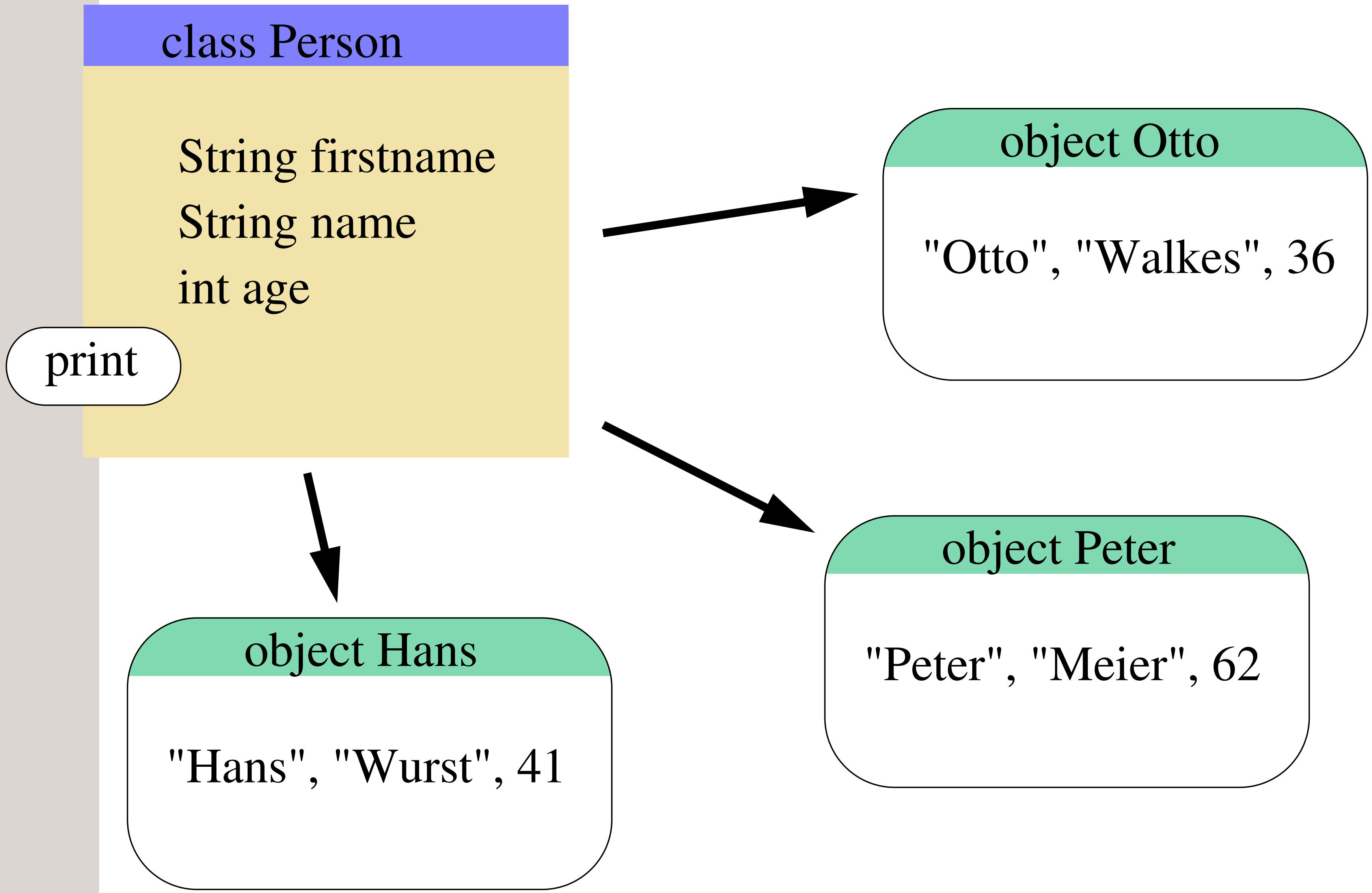
String firstname
String name
int age

print

Class - Example



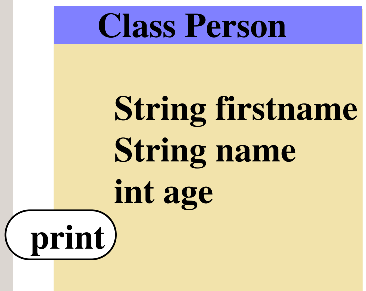
6.5 Class - Example



6.6 Class - Example

- Class definition

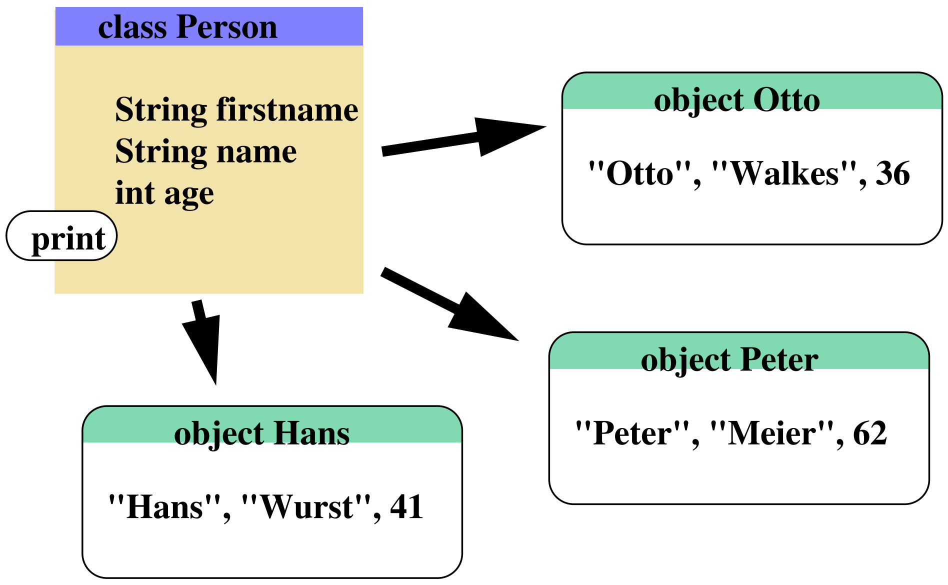
```
class Person {
    String firstname;
    String name;
    int age;
    void print() {
     System.out.println("Name:" + firstname + name +
                        "Age:" + age);
    }
}
```



6.7 Object creation

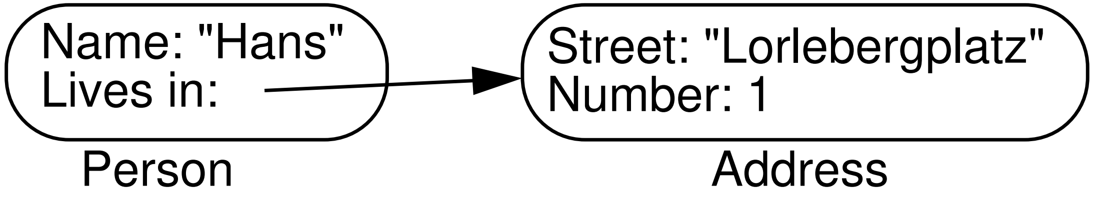
- object creation (instantiation)

```
Person otto = new Person();
otto.firstname = "Otto";
otto.name = "Walkes";
otto.age = 36;
otto.print();
```

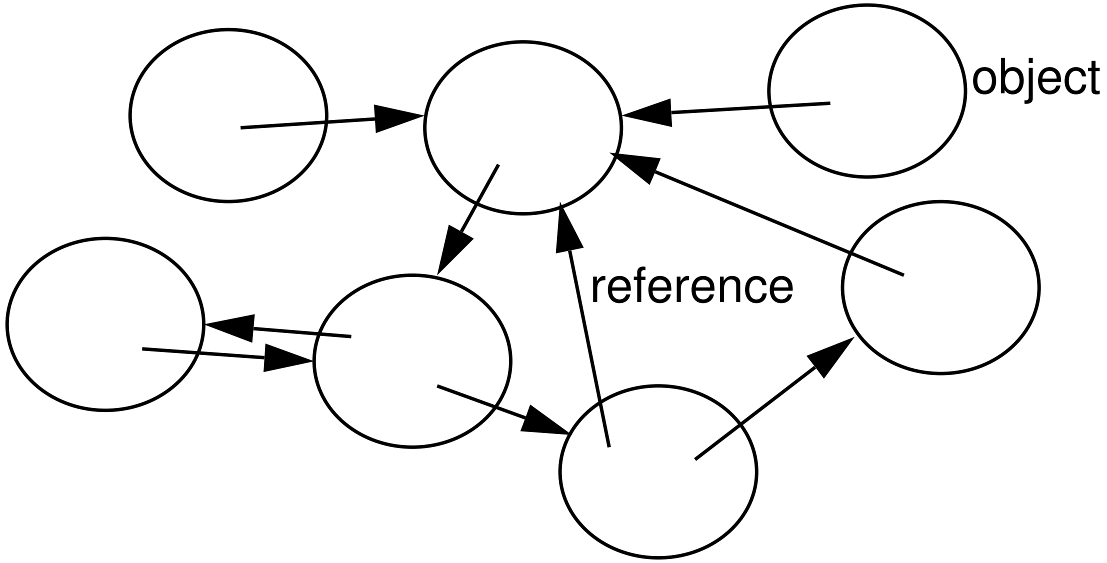


6.8 The object web

- objects can reference other objects
  - ◆ e.g. a person can reference an address



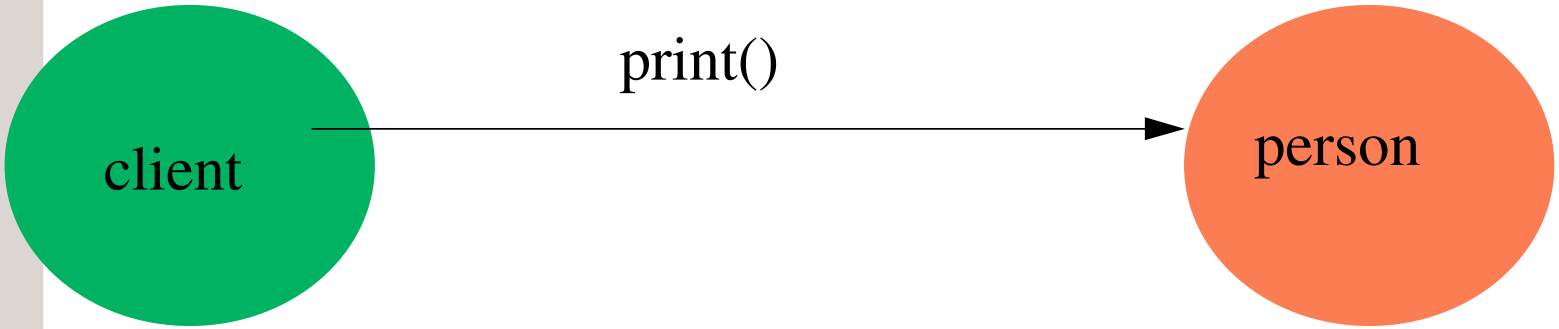
- snapshot of the state of an object-oriented program



6.9 Messages / Methods

- objects communicate through messages
  - ◆ object itself determines its behaviour
  - ◆ object semantics not spread over whole program
- message = method call at an object

```
person.print()
```



- object-oriented program: several objects are communicating to fulfill a specific task

7 Methods and Variables

- Method can access 3 different kinds of variables:
  - parameters (arguments)
  - local variables
  - instance variables

```
class Point {
   int xPosition, yPosition;
   ...
   void move(int x, int y) {
      int i;
      ...
   }
}
```

instance variables

parameters

local variables

7.1 Method-Call Semantics

- object parameters are passed by reference
- primitive types (int, float, etc.) are passed by value

```
void meth(int a, Person k)
    a = 5;
    k.setAge(25);
```

Variables

- Parameters and local variables must have different names
- Parameters and local variables hide instance variables

```
class Test {
    int a;
    void m(int a) {
        a = 5; // does not change instance variable a
    }
}
```

7.3 Variables

- Access to instance variable with
  - *instanceName.variableName*

```
class Person {
    String firstname;
    String name;
    int age;

    boolean sameName(Person otherPerson) {
        if (name == otherPerson.name) return true;
        return false;
    }
}
```

instance variable *instance variable*

7.4 The *this*-Parameter

- every method has an *implicit* parameter **`this`**
- **`this`**: reference to the message-receiver object

```
class Person {
    String name;
    ...
    void print() {
        System.out.println(this.name);
    }
}
```

- **`this`** can be omitted if there are no ambiguities
- Example ambiguity

```
class Person {
    ...
    boolean compare(String name) { return this.name == name; }
}
```

Object Initialization

- creating an object means reserving memory
- this memory must be initialized
- one possibility
  - calling an initialization method
  - bad, because error prone

8.1 Constructors

- initialize the object
- name of constructor = name of class
- are invoked automatically after object creation

8.1.1 Constructors - Example

- Initializing a Person with name and age

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    ...
}
```

8.1.2 Constructors

- more than one constructor possible

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this.name = name;
        this.age = 18;
    }
    ...
}
```

8.1.3 Constructors

- Invocation of a different constructor with **`this(...)`**

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this(name,18);
    }
    ...
}
```

8.2 Direct Initialization

- Instance variables can be initialized directly

```
class Person {
    String name;
    int age = 18;
    ...
}
```

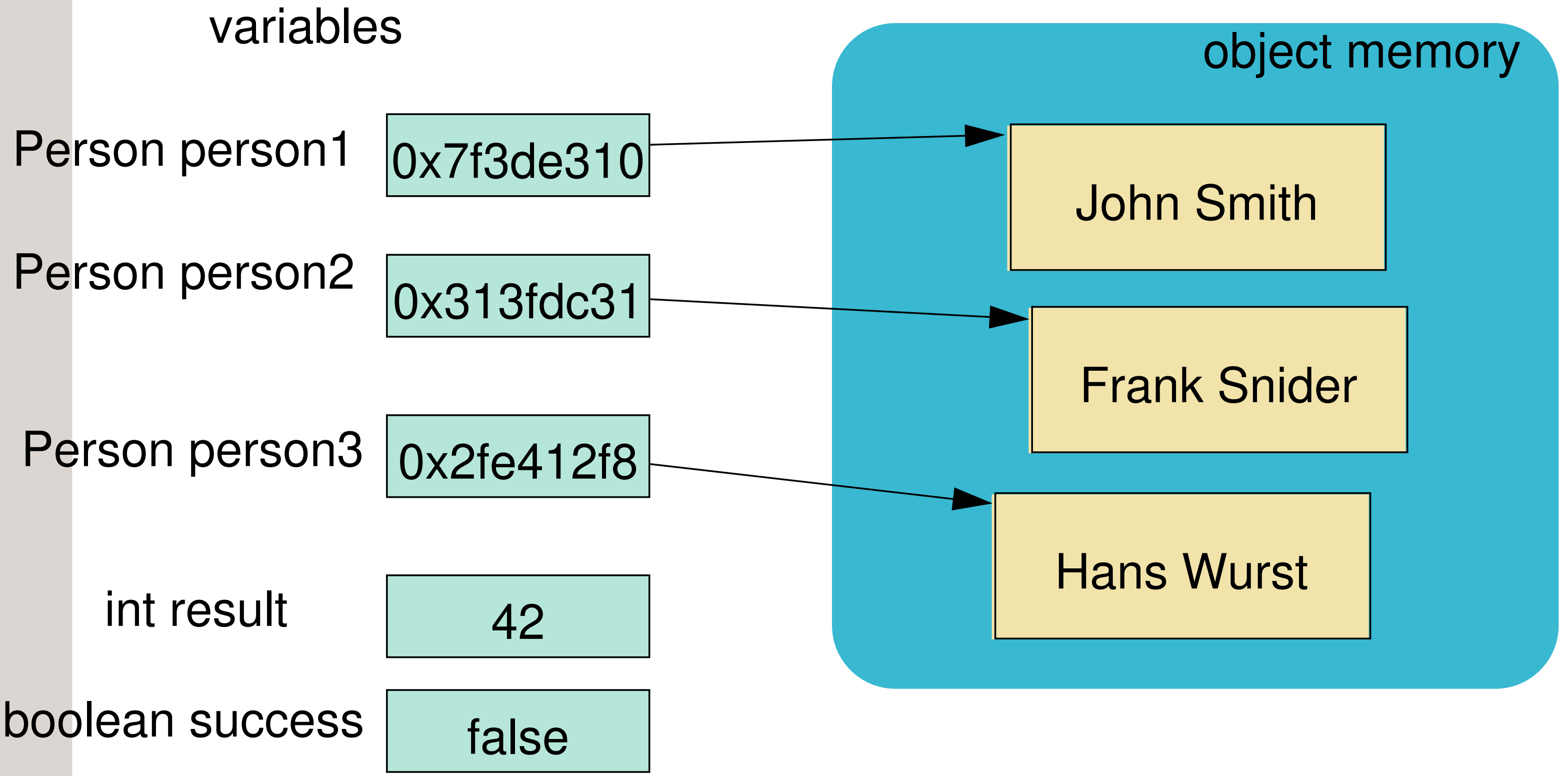
- Initialization is executed at the beginning of every constructor
- Initialization is done in the order of declaration
- Method calls are possible

```
class Customer {
    String name;
    int number = CustomerNumbers.createNew();
    ...
}
```

9 Objects and References

- Java variables do not name objects but references

```
Person p;   // declaration of a reference to an object
            // of class Person
p.print(); // error: method call at the null reference
```



9.1 Objects and References (2)

- = assignes a reference to a variable
- == compares two references
- a variable of primitive type can never be assigned a reference
- a variable of reference type can never be assigned a value of primitive type

```
Person p;   // declaration of a reference variable
int i=42;   // declaration and initialization of
            // a primitive type variable
p = i;      // error: assignment between reference and
            // primitive type
```

10 Equality and Identity

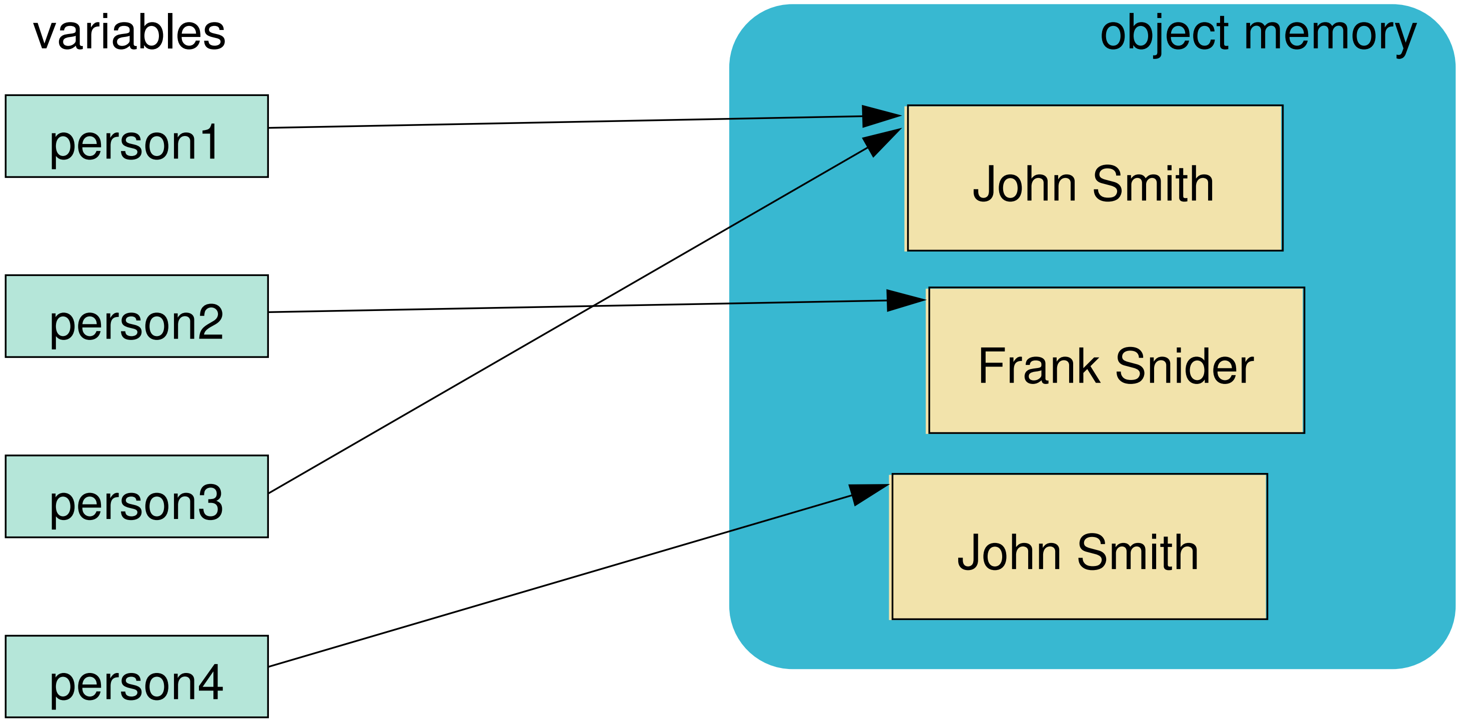
- there is a difference between *equal objects* and *identical objects*

```
class Date {
  int day,month,year;
  Date(int day, int month, int year) {
    this.day = day; this.month = month; this.year = year;
  }
}
...
Date d = new Date(1,3,98);
Date d1 = new Date(1,3,98);
Date d2 = d;
```

- d and d1 are equal
- d und d2 are identical

10.1 Identity and References

- Identity: same reference
- Equality: contents of referenced object is the same



- Which persons are identical, which are equal?

10.2 Test of Equality and Identity

- Identity can be tested using the == operator

```
if (d == d1) { ... }
```

- Equality can be tested calling the **`equals`** method

```
if (d.equals(d1)) { ... }
```

- you have to implement the **`equals`**-method

```
class Date {
    ...
  public boolean equals(Object o) {
    if (! (o instanceof Date)) return false;
    Date d = (Date)o;
   return d.day == day && d.month == month && d.year == year;
  }
}
```

10.2 Test of Equality and Identity

- Java requires: if two objects are equal, then they must have the same hash code

```
class Date {
    ...
  public int hashcode() {
    return day + month + year;
  }
}
```

11 Packages

- Package: program unit (module) with
  - unique name (e.g. **`java.lang`** or **`java.awt.image`**)
  - consits of one or more classes
  - used to partition the name space
- Packages
  - **`public`** classes are available outside the package
  - private classes are only available within their own package
- Visibility of methods/variables can be restricted to the own package
  - Example: **`protected`** = visible only in subclasses or in other classes of the same package

11.1 Classes and Visibility

- A class can be public or not public.
- Classes without the **`public`** declaration are only visible within the same package.

```
public class X { ... }

class X { ... }
```

- A public class must be defined in an own file.
  file name := class name + ".java"
  Example: Class **`X`** must be defined in the file **`X.java`**.

11.2 Keyword: package

- Packages are declared with **`package`**

```
package test;
public class TestClass ...
```

- **`package`** must be the first statement in a file
- you can create a hierarchie of packages

```
package test.unittest1;
```

11.3 Keyword: import

- Classes of other packages can be used with **`import`**:

```
import java.util.*; // use all classes from package java.util
import java.io.File;// use class File from package java.io
```

- classes are searched using the package name as a directory
- Example:
  - Package **`bank`** with class **`Customer`**
  - Use it with **`import bank.Customer`**
  - Bytecode file is searched as **`bank/Customer.class`**
- Package **`java.lang.*`** is imported automatically
- Using classes without import: give full class name, including package

11.4 Packages and the CLASSPATH

- classes are looked up using the CLASSPATH environment variable
- Example:
  - package **`bank`** with class **`Customer`**
  - use this class with **`import bank.Customer;`**
  - compiler,interpreter look for the bytecode file **`bank/Customer.class`**
  - **`CLASSPATH`** contains **`/proj/test:/tmp`**
  - Looking for
    - **`/proj/test/bank/Customer.class`**
    - **`/tmp/bank/Customer.class`**
  - you can specify target directory when compiling
    - **`javac -d /proj/test Customer.java`**

11.5 Package Example

File
`editor/shapes/X.java`

```
package editor.shapes;
public class X {
    public void test() {
        System.out.println("X");
    }
}
```

File
`editor/filters/Y.java`

```
package editor.filters;
import editor.shapes.*;
class Y {
    X x;
    void test1() { x.test();}
}
```

File
`editor/filters/Z.java`

```
package editor.filters;
class Z {
    editor.shapes.X x;
    void test1() { x.test();}
}
```

11.6 Java Core Packages

- `java.lang`: Language support (Thread, String,...)
- `java.io`: Input-/Output support (Files, Streams,...)
- `java.net`: Network support (Sockets, URLs, ...)
- `java.awt`: Abstract Windowing Toolkit
- `java.applet`: Applet support
- `java.util`: Utility classes (Random), data structures (Vector, Stack)
- `java.rmi`: Remote Method Invocation
- `java.beans`: Beans component architecture
- `java.sql`: database access
- `java.security`: cryptographic support

11.6 Java Core Packages

- Look in the documentation to find out more about them!

`http://www4/Services/Doc/Java/jdk-1.1/api/packages.html`

12 Encapsulation

- *Encapsulation* is one of the principles of object-oriented programming
- Encapsulation is used to hide unnecessary information (*information hiding*)

12.1 Encapsulation in Java

- Visibility:
  public, default, protected, private protected, private

- Can be applied to methods and instance variables
  - **`public`**: globally visible
  - default: visible in the same class and the same package
  - **`protected`**: visible in this class, subclasses, and within the same package
  - **`private`**: only visible within the same class

- visibility modifier must be specified with *each* method and instance variable

12.2 Encapsulation

- without encapsulation:

```
class Person {
    public String name;// "name" can be modified/read globally
}
```

- better:

```
class Person {
    private String name;// only Person can access "name"
    public String getName() {// access method
        return name;
    }
}
```

Applications and Applets

- Applications
  - ◆ are started like normal programs
  - ◆ have full access to local resources (disk, network, OS, ...)
- Applets
  - ◆ can be integrated in web pages
  - ◆ access to local resources is limited

13.1 Applications

- An application needs a main method

```
class MyApplication {
    public static void main(String[] args) {
        ...
    }
}
```

- Applications are started by invoking the Java interpreter with the main class (the class containing the **`main()`** method)

```
> set path=( /local/java-1.1/bin $path)
> setenv CLASSPATH /proj/i4oods/golm/assignment1
> java MyApplication
```

13.2 Applets

- Applets are started via an HTML file
- Applets are referenced with HTML tag: **APPLET**
  - Specification of applet class
  - Specification of applet size
  - Optionally some parameters (name and value)

```
<APPLET
    CODE="MyApplet.class"
    width=200
    height=200
>
</APPLET>
```

- **.class** file and **.html** file must be located in the same directory

13.3 Applets

- Class specified in html file must be *subclass of* **`Applet`** (**`MyApplet extends Applet`**)
- Course of events:

1. Applet class (**`MyApplet`**) is loaded.
2. Class is instantiated.
3. Method **`init()`** is called at the **`MyApplet`** object.
4. Method **`start()`** is called.
5. The the applet runs.
6. Method **`paint()`** is called when the applet area must be painted.
7. Method **`stop()`** is called when the applet disappears from the user.
8. When the browser is exited or the applet is removed from the browser cache then method **`destroy()`** is called.

13.4 Applets Example

```java
import java.awt.*;
import java.applet.*;

public class MyApplet extends Applet {
  String s;

  public void init() {
    s = "Hello World";
  }

  public void paint(Graphics g) {
    g.drawString(s, 10, 10);
  }
}
```

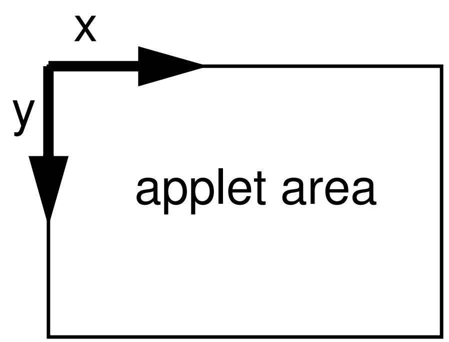
13.4 Applets

- Applets can be tested with netscape (Version# >= 4.06) (/local/netscape/bin/netscape)
- Better for testing: appletviewer
- Invocation with: **`appletviewer test.html`**
  - ◆ you need to include java in your path:

```
set path=( /local/java-1.1/bin $path)
```

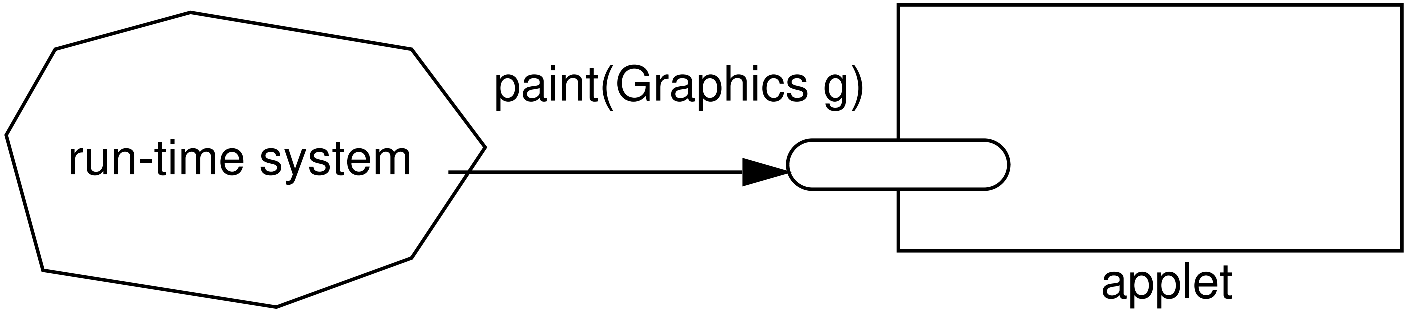
13.5 Mouse Events

- Event handlers of the Applet class
  - **`boolean mouseDown(Event e, int x, int y)`**
    user pressed the mouse button at position (x,y)
  - **`boolean mouseDrag(Event e, int x, int y)`**
    user moved the mouse to (x,y) while pressing the mouse button
- the Event object contains further information about the event
  - check if Shift key was pressed with **`shiftDown()`**
- Return true to indicate that you handled the event
- (x,y) coordinates

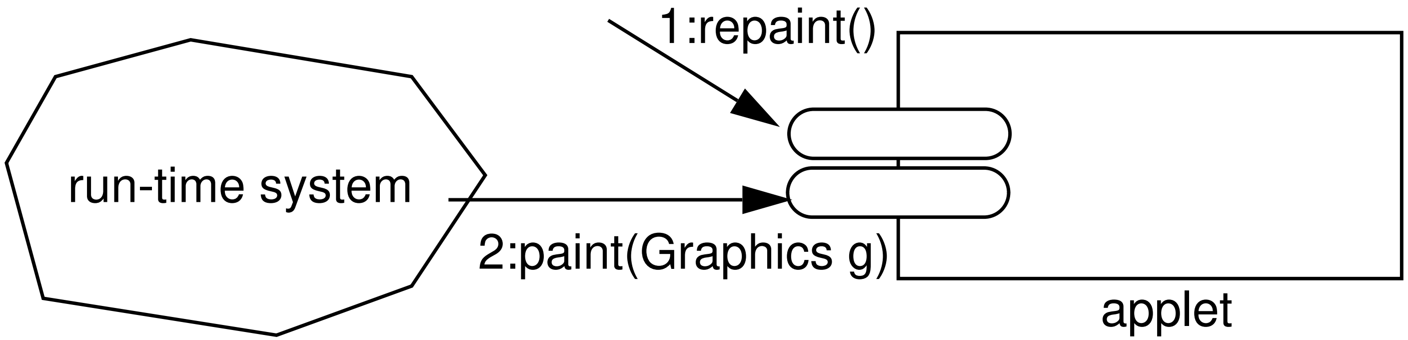


13.6 Applet Painting

- if the Applet must be drawn, the system calls the paint method



- you can tell the system, that paint() must be called by invoking repaint()



13.7 The Graphics Object

- is passed to the paint() method of the Applet as a parameter

```
import java.awt.*;
import java.applet.*;

public class SampleApplet extends Applet {
     ...
   public void paint(Graphics g) {
     ...
   }
}
```

- is used for drawing
  - drawRect(int x, int y, int w, int h)
  - drawOval(int x, int y, int w, int h)
  - ...

13.8 Further Information

- Look at the OODS Homepage:
  `http://www4/Lehre/WS00/V_OODS1/Tutorial/`