Programming I

Based on *Interactive Programming in Java [2]*
Winter 2000

© *Guido Rößling*

University Siegen
Department of Electrical Engineering and Computer Science
Parallel Systems
Hölderlinstr. 3
D-57068 Siegen
Germany

EMail: roessling@acm.org

Contents

General information about Programming I

Contents of Programming I:

- *Introduction to program design*
- *Programs as communicating entities*
- *Specifying program behavior*
- *Programming fundamentals:* expressions, statements
- *Classes* and *objects*
- *Error handling using exceptions*
- *Encapsulation*
- *Inheritance*

General information about Programming I

The *goals* of ”Programming I” are . . .

- ❑ introducing the building blocks of programs,
- ❑ giving an understanding of dynamic systems,
- ❑ enabling you to program (small) applications,
- ❑ teaching the principles of *object orientation*
- ❑ . . . and keeping (or making!) you interested in Programming as a process of *deliberate decisions*, not ”ad hoc” solutions.

Introduction to Program Design

This chapter covers the following topics:

- ❑ What is a *computer program?*
- ❑ What are the *components* of a program?
- ❑ How are the components *put together*?
- ❑ What are the *central design issues*?

What is a Computer Program?

- Computers provide *services* for many different activities
- However, they must be *told* how to provide the service - and be *requested* to do so!
- *Programs* are used to tell computers *how* to provide a given service
- Programs consist of *instructions* in a language the computer ”understands”.
- Program *execution* consist of performing the program instructions.
- A program by itself does no *do* anything: it has to be *executed*!
- The execution can be repeated, but the *result* may depend on certain circumstances such as *user interaction*.

What is a Computer Program?

- One special type of program *always* runs on the computer: the *operating system*
- The operating system is responsible for coordinating what happens in the computer and also handles the starting of other programs
- *Installing* a new program only places a *copy* of the instructions on the computer – similar to placing a new book on a book shelf
- Afterwards, the program can be executed (”**run**”) as often as we want.

Programming Languages

- The instructions understood by the computer must be given in a certain *programming language*
  - ☞ In this course, we will use Java
- *Programming* languages are usually very different from ”natural” languages
- Statements in programming languages must be *unambiguous* and *very precise*
- Usually, we have to specify *each* action precisely - and not gloss over details!
- Instructing computers is (somewhat) similar to teaching young children: be very careful what is done, precise and detailed!

Programming Language ”Example”

Look at the following instructions on how to make a jam sandwich:

1. Get a loaf of bread.
2. Cut off two slices of the bread and put it on a table.
3. Get a package of butter and put it on the table.
4. Get a jar of jam and place it next to the package of butter.
5. Get a knife.
6. Open the package of butter.
7. Use the knife to pick up some butter.
8. Spread the butter on one slice of bread.
9. . . .

Programming Language ”Example”

- The instructions on the previous slide are very specific.
- To make a jam sandwich, the ”user” only has to follow them step by step.
- If *each* instruction is understood by the user, the result of the execution will be a jam sandwich.
- However, if at least one instruction is not understood, it must be rewritten so that it becomes understandable.
- For example, *pick up some butter* might not be specific enough:

  7(a) Place the knife edge parallel to the surface of the butter and lower the knife blade 2 mm
  (b) Move the knife to the other side of the butter package
  (c) Lift the knife blade from the butter package

- Instructions that need further explanation are called *high level*.

Programming Language ”Example”

- The types of statements the computer understands depend on the *specific* programming language
- Some parts are *similar* in most programming languages
  ☞ arithmetics (+, -, ∗, /)
- As a programmer, it is *our* task to make our wishes understandable for the computer
- Therefore, we should often ask ourselves ”*What do I do next?*”

High Level Functions or ”Methods”

❑ Programs are usually grouped into *high level functions*:

1. Assemble the ingredients
2. Spread butter on slice
3. Spread jam on slice
4. Put sandwich together
5. Stow away ingredients
6. Clean up kitchen

❑ In Java , we can also group the operations together in *methods*

❑ Each *method* must provide precise instructions on how to handle the task

❑ Methods may also call other methods.

Programming Primitives

❑ Computers do not ”understand” much on their own - they can mostly *manipulate words and numbers*

❑ Java contains many methods for performing special actions

☞ open a new window, send messages to other computers,...

❑ There are basically three typical *primitive* operations:

☞ Execute commands *in sequence*

☞ Determine next action according to a *condition*

☞ *Repeat* executing operations until a certain condition is *met*

Programming Primitives

- Execute commands *in sequence*
  - ☞ This is what the example covered!
- Make a *choice* (”conditional”):

> 8. **If** top of the slice of bread is covered with butter, go to step 9. **Otherwise**, go to step 7.

- *Repeat* operations (”loop”):

> 7. **Repeat until** the top of the slice of bread is covered with butter:
>
> (a) Pick up a glob of butter
>
> (b) Spread it on the top of the slide of bread

- We will come back to these important parts of building programs later.

Interaction between Programs and Computers

- The example used in the last slides is *autonomous*
  - ☞ it does not have to communicate or interact with others
- However, most current computer programs *interact* with
  - ☞ *people (users),*
  - ☞ *machines,*
  - ☞ *other computers,*
  - ☞ or other *programs* on the *same* computer
- *Interactive* programs are **not** concerned with solving a pre-defined problem and then *stopping*
- Rather, they often perform a *control loop* that responds to *requests*
- Such programs are *embedded* into an environment they *interact* with

Interaction between Programs and Computers

- Running interactive programs is more than simply executing them – they must be *coordinated*
- For example, instead of making a sandwich for *ourselves*, consider having several people running a *restaurant*.
- Typically, the tasks are *shared* by their content:
  - ☞ Take orders and serve food (waiter),
  - ☞ Prepare the food,
  - ☞ Checking and maintaining supplies
- Each task provides *services* to others and *requests* services in return.
- *Coordinating* the tasks requires different questions from simply ”What do I do next?”

Interaction between Programs and Computers

- Let us take a quick look at one of the *tasks*, and we will notice that the ”old” role of sandwiches is still important.
- These are typical instructions for a *chef*:

  1. Pick up a new food order.
  2. Find the instructions for the dish ordered and follow them.
  3. Put the completed dish and the order information on the counter for pickup.
  4. Go back to step 1

- The second step of this ”program” is a *higher level* step as described before
  - ☞ it is not complete, but refers to more detailed instructions
- The computer still follows simple sequenced steps in its execution

Interaction between Programs and Computers

- However, more than *one* process may be active at any time
  - ☞ While the chef cooks a dish, a waiter can take other orders
- The chef's instructions are placed in an *infinite control loop*
- Whenever one order is finished, the chef will look for the next order
- Control loops are *typical* for programs that handle (accept and provide) services
- However, the central question is how we can coordinate the community of programs ("computational community")
  - ☞ how will waiters note that a dish is complete?
  - ☞ how can a client call a waiter?

Coordinating a Computational Community

- Deep down, each program consists of simple instructions
- A single program may have many *entities* following their own set of instructions
  - ☞ think of the staff of the restaurant
- When we look at the whole program, we do not necessarily see the individual steps
  - ☞ We will rather see a *coordinated activity* among the entities
- The *behavior* of the community is *not* the responsibility of *one* member of the community
  - ☞ it is the result of *many* community members working together!

Coordinating a Computational Community

- The *programmer* has to figure out how to tell the computer *what* to do
- There are some fundamental questions that will have to be asked
- For a community of entities, we have to specify
  - the *members* of the community,
  - how *each* member works,
  - and how they *interact*.
- This process is somewhat similar to setting the cast of a play

Central Questions in Designing Communicating Entities

- We have to consider the following question during design:
  - What is the desired *behavior* of the (whole) program?
  - *Who* are the entities who interact to produce this behavior?
  - How does *each* entity work?
  - How do the entities *interact*?
- We will briefly regard these questions now and return to them in more detail later on.

What is the Desired Behavior of the Program?

- ❑ We should address this question *before* we start designing our program!
- ❑ We must ask ourselves…
  - ☞ What *service* should the program provide?
  - ☞ What *guarantees* does the program make about these services?
  - ☞ Under what *assumptions* does to the program make these guarantees?
- ❑ For the restaurant, this might look as follows:
  - ☞ *Provide* clean tables, take orders, serve food, present bill, collect pay
  - ☞ *Guarantee* ”acceptable” waiting time; present appropriate bill; have revenue exceed expenses; have sufficient fresh supplies ready at all times
  - ☞ *Assume* that *customer traffic* is *reasonable* - specify what this means precisely, and what happens otherwise!
    Assume that *more* than one customer can be served at a time.
- ❑ There are more considerations to take into account, but these are ”central”.

Who Are the Members of the Community?

- ❑ We can *not* answer this question in isolation
  - ☞ the answer depends on *what* the entities are and *how* they work!
- ❑ Answer this question in *high-level*, *general* terms, *then* try to address the more precise questions on ”how” and ”what”
- ❑ Normally, we will have to return to this question and rearrange the answers
  - ☞ *incremental program design*
- ❑ In the restaurant, we could define the following entities:
  - ☞ waiter staff unit (for dealing with customers),
  - ☞ kitchen staff unit (for cooking the food)
  - ☞ financial unit (setting of prices, collecting payment, buying supplies)
- ❑ This structure does *not* define how *many* individual entities we will need
  - ☞ Is one waiter enough, or will we need more?

How Does Each Entity Work?

- ❑ Answering this question requires some knowledge about the interaction
- ❑ For each entity, we should ask ourselves. . .:
  - ☞ What are its responsibilites?
  - ☞ What guarantees does it make, and under what assumptions?
  - ☞ What resources does it control?
  - ☞ How does it work?
  - ☞ Is it a community of entities, or a single entity?
    - ☞ Waiting persons might be divided into person(s) who
    - ➪ clear the tables,
    - ➪ take the order,
    - ➪ serve the wine.

How Do the Entities Interact?

- ❑ Here are some basic questions about the interaction:
  - ☞ What are the entities' *interfaces*?
    - ☞ promises made, fulfilled contracts, provided services
  - ☞ How do the entities communicate?
    - ☞ mechanism, interaction pattern, how to keep them alive?
  - ☞ What interaction patterns are possible?
  - ☞ What happens when something goes wrong?
- ❑ The communication is specified by a *protocol*: in the restaurant, a *piece of paper*
- ❑ Some processes must happen in *real time*, others may be performed *in batch*
  - ☞ Food should be served in real time, but turning in the day's payments may happen in batch

The Development Cycle

- Typically, we will be given a set of *specifications* and some *components* to integrate into our new system
- It is helpful to ask ourselves the following questions:
  - What is the *behavior* of the program?
  - Who are the *components* that combine to produce this behavior?
  - How do they *interact*?
  - What is each entity *made of*? (for example a community of entities or a control loop)
- On the level of *instructions*:
  - What happens next?
  - How can this be accomplished?

The Development Cycle

- Once we have answered the questions on on page 2.22, we can start building our program
- We should discuss our design with others: *programmers* and especially our *customers*!
- We may often have to ”go back” and change parts of the specification
  - Make sure to carefully consider what interdependencies might be affected!
- It is helpful to implement only a small piece of the system at first (”prototype”)
  - Stage our design with ”building blocks” that can easily be added
- *Test* our system before we add new features (and afterwards!)
- Produce *understandable* code: use **comments** and **helpful identifiers**

The Interactive Control Loop

- We will focus on *designing interactive software* in this course
- The central piece of this is the *interactive control loop*
- The loop is a simple program that responds to *input*, usually by starting ”appropriate” activities
- While the concept is very powerful, it is also rather simple
- The (probably) simplest interactive control loop is an *echo* program
  - ☞ The program waits for input and simply *echos* the input back
- While extremely simple, this program exhibits many important properties:
  - ☞ it is embedded in an environment (the typing and display)
  - ☞ it is interactive – it could also send the output to another user!),
  - ☞ it is *concurrent* (the user may continue typing while the line is printed)

A Community of Interacting Entities

- In the following, we will take a look at *transformations* on *words* and *phrases*.
- More precisely, we will build a system which can contain many transformers that can be *combined* in almost any order
- As the underlying example, we use *Pig Latin*
- *Pig Latin* is just plain English with some modifications put in
- Example: place the *first* letter of each word at the *end* and then add ”`ay`”:
  `"Hello, how are you"` ⇨ ”`ellohay, owhay rehay ouhay`”
- There are *many* applications including
  - ☞ *capitalizers*: `hello` ⇨ `HELLO`
  - ☞ *name givers*: `hello` ⇨ `Guido says "hello"`

A Community of Interacting Entities

- There are two further special transformer types: *collectors* and *repeaters*
- *Collectors* listen to multiple sources, but produce only *one* (usually combined) output
  - ☞ used when many people try to ”talk” at the same time
- *Repeaters* produce more than one output from one input
  - ☞ used when we want to broadcast a message to others
- Our system will contain many *transformer* ”boxes” shown in figure 1

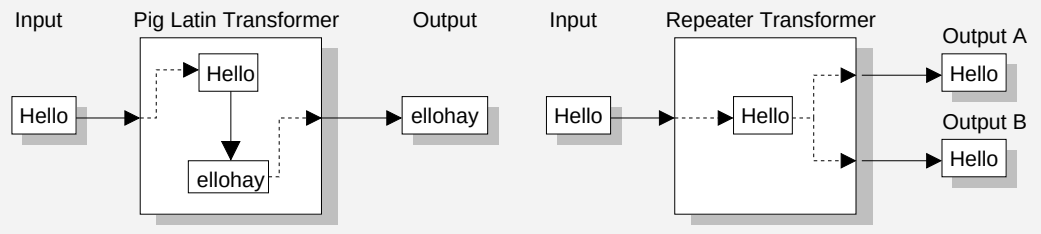

Figure 1: Transformer principle

A Community of Interacting Entities

- At the end of the design process, we should be able to provide a scenario for *each* of the major interactions with the system
- This includes
  - ☞ what roles have to be filled (the types of things in the system),
  - ☞ who fills these roles (the individual objects)
  - ☞ how these objects communicate among themselves (flow of control)
- Let us view the system as a community of *interacting transformers*
- Each transformer then will be an entity
- The interaction consists of *requesting* input and *providing* (transformed) output
- We want to be able to combine all transformers
  - ☞ Output of transformers A must be usable as input for transformers B

A Community of Interacting Entities

- We need to connect transformers so that input / output can be reused
- A *connection* simply lets us insert something at one end, and lets someone receive it at the other end

Transformer entity interactions, version 1:

- *Read* a word or phrase from a *connection*
- *Write* a word or phrase to a *connection*

Connection entity interactions:

- *Accept* a word or phrase written on the connection
- *Supply* a word or phrase when requested

The User and the System

- Let us consider the interactions between a *user* and the system:

  System/User interactions:

  ☞ *Create* a transformer (of a specified type)

  ☞ *Connect* two transformers (in a particular order)

- To support this, we add a *user interface* with a *control panel* of *buttons*
- The user interface supports the creation of a specific transformer
- By selecting *two* transformers, they will be connected in the order they were selected using a new *connection*

  User Interface interactions:

  ☞ *Create* a transformer (of a specified type)

  ☞ *Connect* a *connection* between two transformers

❑ We also have to update the *transformer* entity transactions:

Transformer entity interactions, version 2:

- ☞ Accept an *input connection*
- ☞ Accept an *output connection*
- ☞ *Read* a word or phrase from a *connection*
- ☞ *Write* a word or phrase to a *connection*

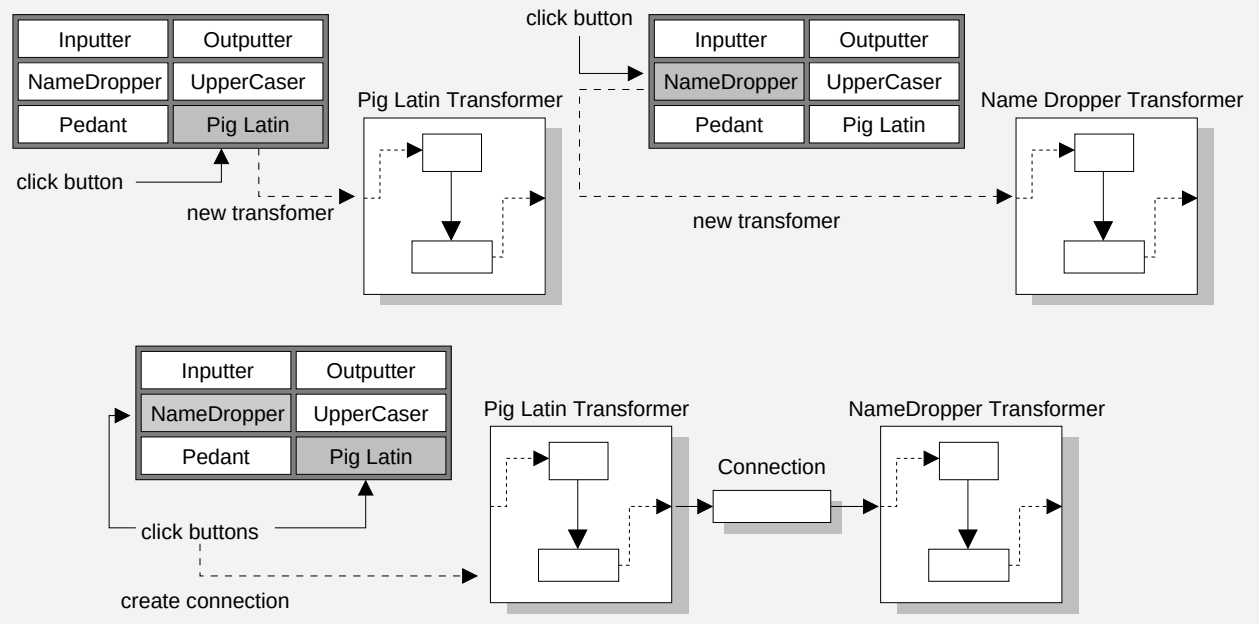

Figure 2: Diagram of the Transformer User Interface

Control Flow: Creating a Transformer

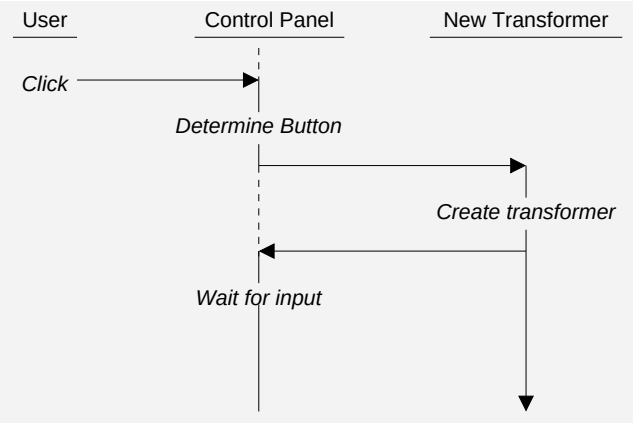

Figure 3: Control flow when creating a transformer

Control Flow: Creating a Connection

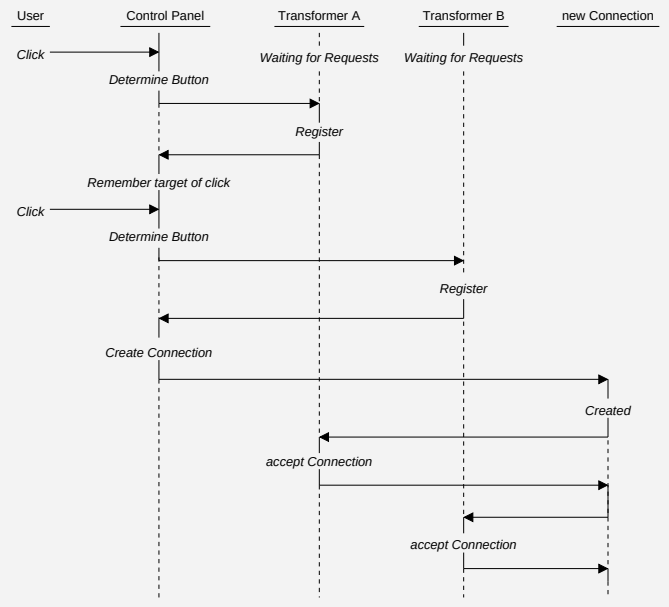

Figure 4: Control flow when creating a connection between transformers

What goes Inside?

- The last diagrams showed the *interactions* between the entities
- We have also addressed *who* the community members are
- The *control flow* diagrams show the interaction
- We may have an arbitrary number of *transformers* and *connections* in the system
  - For example, 5 transformers and 2 connections
- However, there will be only *one* control panel
- Next, we would have to check the entities whether they are composed of other entities
- However, we still lack some background for this
  - the rest of the chapter will look instead *transformers* instead

Building a Transformer

- A Transformer must be able to
  - *Accept* an input connection
  - *Accept* an output connection
  - Have an independent ”agent” that reads input, transforms it, and writes the output
- Therefore, the transformer is a community of entities:
  - two *connection acceptors* which are activated on a connection accept request
    - they will have to remember the connections that they accept
    - they must also decide what to do if a connection is requested more than once
  - an *input transformer* that will *read* from the input connection, performs the transformation, and writes the result to the output connection

Transformer Examples

- Here are the instructions for the behavior of Capitalizer transformers:

Capitalizer
1. Read from the input connection
2. Produce a capitalized version of it
3. Write this to the output connection

- NameDroppers have to store their own *name*:

NameDropper
1. Read from the input connection
2. Produce a phrase containing *your name* followed by `says '`, the input, and `'`
3. Write this to the output connection

Transformer Examples

- Repeaters send the *unmodified* output to *two* output connections:

NameDropper
1. Read from the input connection
2. Write this to the *first* output connection
3. Write this to the *second* output connection

- The **general** instruction form for *any* Transformer is

NameDropper
1. Read from the input connection
2. Produce a transformed version of the input
3. Write this to the output connection

- We will now look at the second instruction.

Strings

- *Strings* are special Java objects for representing *words* and *phrases*
- A Java String can be *anything* that is placed between a pair of double quotes `"..."`
- The *double quotes* are used to mark the *start* and *end* of the *String*
  - ☞ they do *not* belong to the phrase
- Some legitimate Java Strings are `"Hello", " Spaces "m "!&$#!"`
- The *transformers* are really *StringTransformers*
  - ☞ each transformer takes a String and produces a *String* as its output
- In the following, we will look at some of the predefined Java String transformations

Fundamental String Operations

- Strings are concatenated (put together) using a plus `+`
  `"Hello, I'm " + "Guido"` ⇨ `"Hello, I'm Guido"`
- Thus, a NameDropper can produce its output as follows, assuming the name is stored as *myName*:
  myName + " says '" + *input* +"'"
- There are many *methods* that can be performed on *Strings*, such as transforming a String into upper case
- Using such an operation is called *invoking* it
- To *invoke* a *method* on a *String*, put a dot `.` between the *String* and the *method name*:
  "Hello".toUpperCase() ⇨ "HELLO"

Fundamental String Operations

- A String can return a *substring* of itself
  - The characters are numbered starting with *0*
  - For retrieving a substring *starting* at position 3, *invoke*
    `"Hello".substring(3)` ⇨ ”lo”
  - For retrieving the middle three characters of ”Hello”, *invoke*
    `"Hello".substring(1, 3)` ⇨ ”ell”
- `toUpperCase()` can be used to produce an upper case text:
  `"MixedCase".toUpperCase()` ⇨ `"MIXEDCASE"`
- `toLowerCase()` produces a lower case text:
  `"MixedCase".toLowerCase()` ⇨ ”mixedcase”

Fundamental String Operations

- *trim()* removes *leading* and *trailing white space* (spaces, tabs. etc):
  `" lots of space ".trim()` ⇨ `"lots of space"`
- *substring(fromIndex)* produces a String starting at position *fromString*, *substring(fromIndex, toIndex)* contains all characters until *toIndex - 1*:
  - `"Hello".substring(3)` ⇨ `"lo"`
  - `"Hello".substring(1, 4)` ⇨ `"ell"`
  - `"Hello".substring(0, 5)` ⇨ `"Hello"`
- `length()` returns the number of characters in a String:
  `"Hello".length()` ⇨ 5
  - Note that the *last character position* is at `length() - 1`
- `replace(old, new)` replaces all occurrences of `old` by `new`:
  `"Hello".replace('l', '*')` ⇨ `"He**o"`

Fundamental String Operations

- `charAt(pos)` returns the character at position `pos`

```
"Hello".charAt(2) ⇨  'l'
```

  - ☞ For a valid invocation, make sure that $0 \leq$ `pos` $<$ *String*.`length()`

```
"Hello".charAt(-1), "Hello".charAt(5) ⇨ Error
```

- `indexOf(char)` returns the *first* index of the character or `-1`

```
"Hello".indexOf('l') ⇨  2
"Hello".indexOf('h') ⇨  -1
```

- `lastIndexOf(char)` returns the *last* index of the character or `-1`

```
"Hello".lastIndexOf('l') ⇨  3
```


Rules and Methods

- The String operations introduced can be used to construct the instructions needed for a *Transformer:*

```
to transform a String (say, thePhrase),
  return thePhrase.toUpperCase();
```

- This rule describes the transformation rule for an UpperCaser
- The transformation requires a String that is to be transformed
- Within the rule, this String is addressed as `thePhrase`
- `thePhrase` is called the *parameter* of the rule
- A *concrete* value for `thePhrase` is called an *argument*
- the *name* of the parameter can be chosen arbitrarily
- `thePhrase` can be replaced by *any* String to be transformed

Rules and Methods

- The *rule* or **method** from the previous slide is not valid Java code
- However, we only have to change it *slightly* to make it correct Java!
- The ”correct” method for *UpperCaser* looks like this:

```
String transform(String thePhrase)
{
  return thePhrase.toUpperCase();
}
```

- The `String` at the beginning of the method is the *return type*
  - ☞ Thus, the method *takes* a `String` argument and returns a `String`
- This *method* can be used by *anybody* with an *arbitrary* `String`.

Classes and Instances

- The *method* just described is the *only* thing that distinguishes the *Transformers* from each other
- It is useful to make each *Transformer* a new **type**
- **Types** consist of a *name*, *method* and internal *values* (called **attributes**)
- In Java (and other object-oriented languages), a type is called a **class**
- A Java **class** is started with the *keywords* **public class** followed by its name
- The class may *extend* another class - in this case, let us call the basic class `StringTransformer`
- The class body simply contains the *methods* and *attributes*, placed between curly braces { }

Classes and Instances

- The definition of the appropriate *class* for *UpperCaser* looks as follows:

```
public class UpperCaser extends StringTransformer
{
  String transform(String thePhrase)
  {
    return thePhrase.toUpperCase();
  }
}
```

- The code for a *Pedant* is very similar:

```
public class Pedant extends StringTransformer
{
  String transform(String thePhrase)
  {
    return "Obviously " +thePhrase;
  }
}
```

- How can we specify the other transformers from this example?

Classes and Instances

- These *classes* are descriptions of what a *Transformer* should do
- However, they are **not** *UpperCasers* or *Pedants* themselves
- Rather, they are like a "recipe" or *template* from which a *particular* Transformer can be made
- To turn the class into a "real" transfomer, we use the **new** command:

  new `UpperCaser()` or **new** `Pedant()`
- These operations will return a *particular* instance called an **object**
- If we use the **new** command again, *another* object will be generated
- This is exactly what the GUI buttons are supposed to do (see slide 3.7):
  - ☞ Pressing the button labelled `Pedant` will invoke **new** `Pedant()`, generating a new `Pedant` object

Classes and Instances

- A *UpperCaser* can be connected to an *Pedant* through the GUI
- What will happen if the *UpperCaser* input is ”not much”?
  - The *UpperCaser* produces the output ”NOT MUCH”
  - The *Pedant* prepends ”Obviously”: ”Obviously NOT MUCH”
- If we pass this output into *another* Pedant, it will output ”Obviously Obviously NOT MUCH”
- Thus, the output of a *Pedant* will always start with *at least one* ”Obviously”

Fields

- How can be realize the slightly more complex class *NameDropper?*
- The *NameDropper* is supposed to add its *name* followed by `says '`, the phrase, and the final `'`
- Thus, *my NameDropper* should output *”Guido says ’Hi!’”* on input *Hi!*
- The rule looks like this:

```
to transform a String (say, thePhrase),
   return myName +" says '" +thePhrase +"'";
```

- However, *myName* is **not** a parameter – it is a *persistent* part of the *specific* NameDropper!
- Thus, *each* NameDropper must be supplied with a *name* when it is *created*
- The *given name* must be *stored* so that it can be used when needed

Fields

- To *store* the name of a *NameDropper* object, we need a *storage space*
- In Java, these are called *fields* or *attributes*
- In this case, we will call the field `name`
- The *type* of the field will be `String` ☞ why
- To make clear that we want to refer to *the object's own name* in the method, we can use `this.name`
- `this` always refers to the *current object*
- The *NameDropper* method thus looks as follows:

```
String transform(String thePhrase)
{
  return this.name +" says '" +thePhrase +"'";
}
```

- Of course, this method has to be embedded in a class, too!

Fields

- A typical example for *NameDropper* usage is shown in figure 5

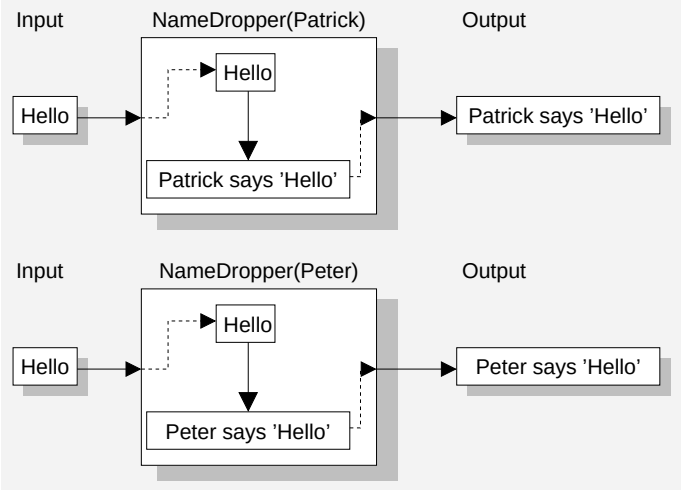

Figure 5: *NameDropper* example

Fields

- We need a place to store the *name* used in a *NameDropper*: `String name`
- The *constructor* that builds a new object should now ask for a *target name*
- Basically, the rule for the constructor is

> to construct a `NameDropper` with a `String` (say, `targetName`),
> *assign* `name` the value of *targetName*;

- In Java code, the method will look as follows:

```
public NameDropper(String targetName)
{
  name = targetName;
}
```

- To generate a new *NameDropper*, we have to supply a *String* parameter:

```
new NameDropper("Patrick");
```


Fields

```
public class NameDropper extends StringTransformer
{
  String name;

  public NameDropper(String targetName)
  {
    name = targetName;
  }


  String transform(String thePhrase)
  {
    return this.name + " says '" +thePhrase + "'";
  }


  public void setName(String myName)
  {
    name = myName;
  }
}
```

General Nature of Approach

- The examples rely on the existence of a *generic* StringTransformer class
- This class must know how to...
  - accept an *input connection* (and store it in a field),
  - accept an *output connection* (and store it in a field),
  - how to create an individual *StringTransformer*,
  - how to invoke the `transform` methods of specific StringTransformer instances over and over again.
- The `StringTransformer` class is similar to the examples we have seen, but somewhat larger
- All transformers we have seen obey the same general *rules* and *interfaces*
  - Each defines a `String transform(String)` method
- Thus, we can always speak of "a transformer", without having to worry about the *actual* type of the transofmer.

Objects, Types and Names

- This chapter addresses the following questions:
  - What kind of "things" can computers talk about?
  - How can we keep track of the "things" we know about?
- The objectives include that we will be able to...
  - recognize Java types,
  - distinguish Java primitives from object types,
  - declare and define *variables*,
  - understand declarations as fixing types to a name,
  - recognize that each name contains *exactly* one value
  - understand how a name can refer to something or to nothing,
  - tell when the values associated with two names are *equal*.

Primitive Types and Literals

- Like most programming languages, Java has some *built-in* ways of handling simple types
- This includes *numbers*, *characters* and *truth values* (`true, false`)
- For example, Java knows that the value `6` placed in source code refers to an integer value.
- `6` is called a **literal**: an expression that Java understands "literally"
- Thus, the following are Java *literals*:
  - `42`
  - `-3.7`
  - `'a'`
  - `true`
- *Single* quotes are used to mark `'6'` as a *character*, not a *numeric value*.

Boolean Values

- Java supports the *boolean values* (or "truth values") `true, false`
- These are helpful for manipulating *decisions*
- *Boolean* values are commonly used in *conditional* instructions
- There are only **two** literal values for *booleans:* `true` and `false`
- Note that Java is *case-sensitive*: `true` is **not** the same as `True`!

Strings

- As we saw in the last section, Java also offers some support for *Strings*
- Some typical uses for *String* objects are
  - error messages,
  - prompting the user for input,
  - window titles,
  - messages sent to the user.
- Strings are surrounded by *double quotes* "…"
  - If we want to use a double quote *inside* a *String*, use \"

Objects

- The primitive types are very specific, and thus limited in their functionality
- Most of what Java does therefore works with more complex types called *objects*
- For example, think of the elements of the graphic user interface – they are far more complex than a simple *number*!
- Java has a simple rule: anything that is *not* a primitive is an *object*
- There are lots of different object kinds: *buttons, windows, lists, …*
- We already know several object types:
  - the *StringTransformer* objects
  - *Strings* are also objects

Objects

- Each object has a number of *services* it can provide
- Two objects of the same *type* offer the same *services*
- However, the *results* may differ depending on *parameters* and the *object state*
- Some objects act on their own
  - *Animators* may paint a series of pictures every $\frac{1}{30}s$
  - A *clock* will continue running after it has been started
- Objects are thus far more interesting than primitive types
- However, they are also more *complex*
- Furthermore, there are no *literal* values for objects (except for *String* literals)
- Nearly *everything* we will do in Java deals with objects

Naming Things

- We will need some way to *keep track* of the primitive values and objects used in a program
- This is accomplished by giving them a *name* – a ”name assignment”
- Naming a thing is similar to sticking a label on it
- Afterwards, the name is *bound* to the value associated with it
- The Java syntax for assigning a value to a name is easy:

```
name = value;
```

  - For example, `nrWheels = 4;`
- This associates the value `4` with the name `nrWheels`
- Using `nrWheels` in the program is now the same as using `4`
- Several *names* can refer to the *same* value
- However, each *name* will always refer to only *one* value

Types

- Until now, we were rather casual about our ”things”
- Java is *strongly typed* – which means it is *far* from casual about *what* a thing is!
- Each Java ”thing” has a *type* stating what kind of thing it is
- Java names are created with a *type* and can only label objects of this type
- Before *using* a name, we have to *declare* it as being of a certain type
- This declaration states that the particular name can be used for labeling primitives or objects of only the *particular* type
- Java declarations have the form ”type-of-thing name-of-thing ;”:

```
int nrOfWheels;
boolean canDrive;
String brandOfCar;
```

- Some names are *reserved* and may *not* be chosen – see 4.20

Types

- Java is **case-sensitive:** a and A are *different* words
- We can use an arbitrary amount of *white space – spaces, tabs, line breaks* – between elements, but **not** in names.
- Note that the label sticks to the object
- Thus, even if we change *internal data* of the object, the name will still stick to it

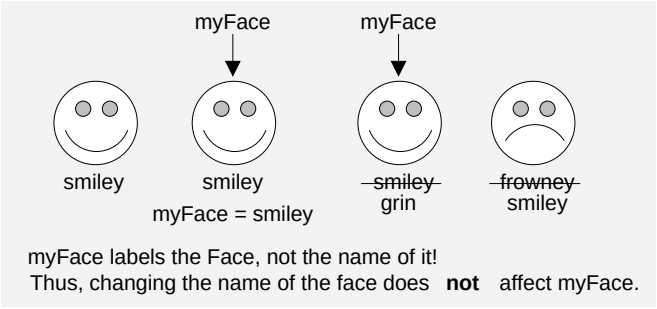

Figure 6: Labels stick to objects, not their internal data

Definition = Declaration and Assignment

- *Declaring* a name requests that Java reserves storage for the object
- However, we will often want to *immediately* assign a *value* to the name
- This *defines* the name's *type* and *value* in one step
- To do so, add a standard *assignemt* immediately after the declaration (before the semicolon `;`):

```
int i = 42;
boolean amIHappy = true;
Face smiley = new Face(amIHappy);
Face frowney = new Face(false);
public int round(float f);
```

- The first two *definitions* use *literals*, while the others actually create new objects - one happy and one unhappy face

Primitive Types

- Java supports the following primitive types:

Type	Description	Range	Default value
boolean	Truth values	`true, false`	`false`
char	16 bit Unicode character	$'\backslash u0000' \ldots '\backslash uFFFF'$ or keyboard chars ('x')	$'\backslash u0000'$
byte	8 bit integer	-128 ... 127	0
short	16 bit integer	-32768... 32767	0
int	32 bit integer	$-2^{31} \ldots 2^{31}-1$	0
long	64 bit integer	$-2^{63} \ldots 2^{63}-1$	0
float	32 bit real number	$\pm 3.4028E+38 \ldots \pm 1.4024E-45$	0.0
double	64 bit real number	$\pm 1.79769E+308 \ldots \pm 4.94066E-324$	0.0

Table 1: Java Primitive types and their range

- Note that the names of the primitive types are *reserved* in Java

Special Character Literals and Umlauts

❑ Table 2 shows the special characters usable in Java

Character	Interpretation	Character	Interpretation
'\b'	*Backspace*	'\f'	*Form Feed*
'\n'	*Newline*	'\r'	*Carriage Return*
'\\'	*Backslash*	'\''	Apostrophe / *single quote*
'\"'	*Double quote*	'*ddd*'	Octal ($0 \leq d \leq 7$), '\000' ...'\377')
'\u*XXXX*'	Unicode character, $0 \leq X \leq$ F	'\u00C4'	German umlaut Ä
'\u00D6'	German umlaut Ö	'\u00DC'	German umlaut Ü
'\u00E4'	German umlaut ä	'\u00F6'	German umlaut ö
'\u00FC'	German umlaut ü	'\u00DF'	German umlaut ß

Table 2: Special Character Literals

Object Types

❑ Java contains a large number of *predefined* object types

☞ For example, think of the *String* class we have used!

❑ We will focus on how we can define new object types and generate objects in the next sections

❑ By convention, the name of object types *always* starts with a *capital letter*

❑ Note that we may *not* use special characters when defining a new object

❑ Each individual object has a number of *properties* and a *behavior*

❑ Objects "do" something when we *request* a service or *invoke a method*

❑ The typical notation for invoking a method is

`nameOfObject.methodName()`

```
UpperCaser upperCaser = new UpperCaser();
upperCaser.transform("Hello World!");
// output will be "HELLO WORLD!"
```

Example Object: `System.out`

- `System.out` is a special Java object that prints mesages to the terminal
- `System.out.print(aString);` prints the String
- `System.out.println(aString);` prints and then starts a new line
- Thus, the output for the following code will be `A is for 'Apple'`:

```
System.out.print("A");
System.out.print(" is for 'Apple'");
```

- However, the following code will instead produce `A is for 'Apple'`:

```
System.out.println("A");
System.out.println(" is for 'Apple'");
```

- `System.out` is very helpful for printing out information about our program - the simplest form of interaction!

Types of Names

- Each Java name has a *type* associated with it
- We can *not* change the type associated with a particular name!
- There are two different kinds of names in Java : *value names* and *reference names*
- When we declare a name as a *value*, a *space* is reserved for holding the appropriate value
- `int i` thus associates `i` with a storage for a 32 bit integer
- We must *assigned* a *value* before we can *read* the variable!
  - ☞ (For advanced programmers: Java does **not** initialize local variables!)
- Assigning a value replaces the current value with a *copy* of the appropriate value

Types of Names

- What happens in the following *definitions*?

```
int i;
i = 42;
int j = i;
i = 54;
```

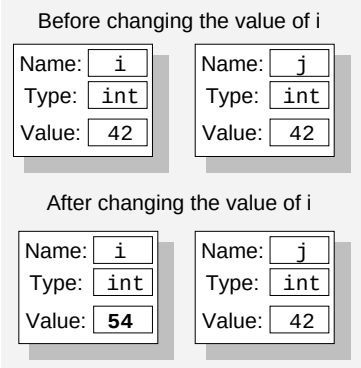

Figure 7: Primitives contain a *copy* of the values

Reference Names

- *Names* associated with *objects* are *reference names*
- The name can then be used to easily refer to a *given* object
- The same "*type-of-thing name-of-thing*" form for declaration is used:

```
String demoString1;
```

  creates a label `demoString` for sticking on a *String* object
- However, the name is *not* stuck on anything!
- Having a *label* is *not* the same as having something to *stick it to*!
- If the label is not *stuck* to anything, it has the special "non-value" `null`
- To avoid errors in our coding, *always* replace the declaration above with either of the following:

```
String demoString2 = null; // no value yet... show by setting to null
String demoString3 = "Hi!"; // known value, so assign it directly
```

Interlude: `JRadioButton`

- For the next example, we introduce a new type: `JRadioButton`
- Objects of type `JRadioButton` are used for buttons that can be *selected* or *deselected*
- Typically, a `JRadioButton` is generated with a *displayed title*
- The displayed title is set in the **new** invocation and stored internally
  - ☞ Recall how we did that in the `NameDropper` class!
- The label of a button can also be *changed* by calling the method

  `setLabel(String newLabel)`
- Calling this method affects the *internal* value of the button
  - ☞ The label of *all* references to the current button is changed!

Interlude: `JRadioButton`

- What happens in the following code?

```
JRadioButton button1, button2; // reserve storage space
button1 = new JRadioButton("Hi!"); // instantiate first button
button2 = button1; // let button2 refer to the same object
button1.setLabel("Hello!"); // change the internal label
```

- What happens is that the label *refers to* the associated value:

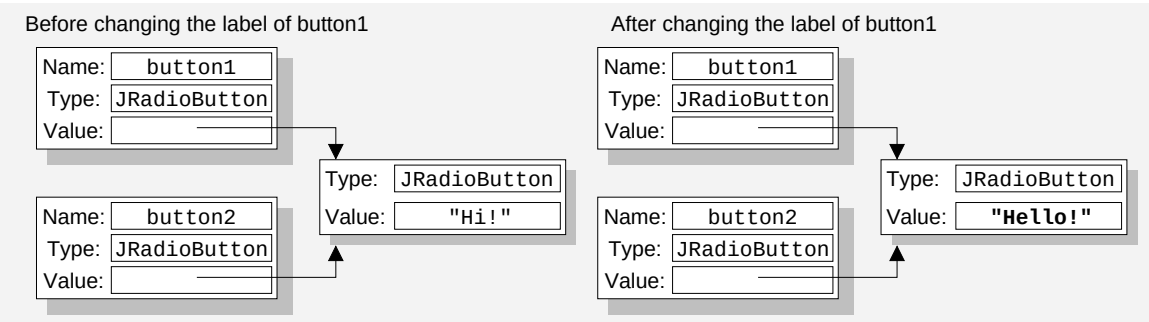

Figure 8: Changing an object referenced by two labels

Reserved Keywords

- Java names ("identifiers") must follow the following rules:
  - The first character must be *letter*, $ or _
  - All following characters must be *alphanumeric* (letters or digits), $ or _
- The *reserved* names ("keywords") hold a special meaning in Java can thus *not be used* as names:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

Table 3: Java keywords

A Note on Equality

- When are the values associated with two names *equal*?
  - Two *value names* are *equal* if they have the same value
  - Two *reference names* are *only* equal if they *refer to the same object*
  - All Java objects provide a method

    ```
    boolean equals(Object other)
    ```

    that compares the current object with the parameter
  - Thus, there are *two* tests for equality:
    - Check for *same reference* or *primitive value*: `==`
    - Check for *equal content of objects*: `equals(otherObject)`
  - The `equals` method will **always** return **true** if both refer to the same object
  - In most cases, the method will *also* return **true** if *all primitive or object values* of both object are *equal*

A Note on Equality

```
int i = 42;
int j = 42; // equal to i: value name, same value
System.out.println("i==j ? " +(i==j)); // true
j = 43; // not equal to i: j (==43) != i (==42)
System.out.println("i==j ? " +(i==j)); // false
JRadioButton button1, button2;
button1 = new JRadioButton("Hi!");
button2 = button1; // equal: same referenced object
System.out.println(button1 == button2); // false(!)
System.out.println(button1.equals(button2)); // true
button2 = new JRadioButton("Hi!"); // MAY be equal
System.out.println(button1.equals(button2)); // true
// however, this depends on how the 'equals' method is implemented
```

- The *last* operation depends on the implementation of the `equals` method
- For `JRadioButtons`, the method will return `false`
  - the two buttons *refer* to different objects that just have the same *label*
  - However, their *behavior* could be completely different!

Specifying Behavior: Interfaces

- We will now examine the following questions:
  - How do *programs* (and *people*!) know what to expect of an object?
  - How can we describe an entity's part or *property* to other *community members*?
- By the end of the chapter, we should be able to...
  - recognize and read Java method signatures
  - understand how an *interface* specifies a *contract* between two entities
  - see how an interface separates the *user* from the *implementation*.

Interfaces Are Contracts

- Programs are *communities* of *interacting entities*
- **But:** how deos one entity know what *services* another entity provides?
- How can we as *programmers* know the behavior of objects we have not programmed ourselves?
- *Interfaces* are a key to understanding these questions
- An interface is a **contract between entities**
- The interface represents an agreement between the *implementor* of an object and its *users*
- The interface specifies *required behavior*
- However, it does *not* necessarily say **how** the behavior is implemented
- The contract does not forbid the user from *adding other operations*

Example Interface: Electrical Outlets

- *Electrical outlets* are a *standardized interface*
- The *shape*, *size* and *electrical properties* of wall outlets are standardized
- Therefore, *any* electrical appliance can be plugged in and will work
- The user can not tell *how* the electricity was produced – by wind, water, . . .
- *However,* there is *more* than only one outlet interface:

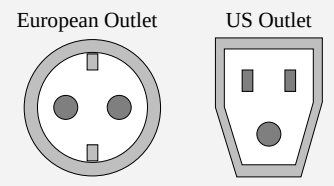

Figure 9: Power outlet interfaces - (Continental) European and US

- If you want to use *European* appliances in the US, you need an *adapter*
  - ☞ The same is true for software components!

Generalized Interfaces and Java Interfaces

- A dictionary defines *interfaces* as "the area common to two or more systems" [1]
- In *Computer Science*, we use interfaces to mean the *boundary* between two (or more) *entities*
- In general, in constructing a community of interacting entities, the interface is the "*face*" these entities show to each other:
  - ☞ what *services* are provided?
  - ☞ what information is *expected* when requesting a service?
  - ☞ what information is *returned* after a service (if any)?
- Therefore, *interfaces* are central to the question of interaction.

User Interfaces

- *User Interface* refers to the the part of a program that the user actually interacts with
- Often, this is a *Graphical User Interface (GUI)* using windows, menus and buttons
- Good user interfaces take into account both the properties of the program and of the user
- It is therefore important that we try to design the interface to fulfill the needs of the entities on *both* sides of the keyboard

Java Interfaces

- Java has a related, but rather more limited, use of the word *interface*
- A Java interface is a particular *formal specification* of the object behavior
- The keyword `interface` is used to specify the declaration of a contract for this behavior
- Java defines the rules for setting out the contract
- However, some interesting properties can *not* be expressed in Java interfaces
  - ☞ The resources a calculation needs, the valid parameter range, ...

An Example Java Interface

- Think of a *counter* as those used on many WWW pages
- The basic operations of such a counter are
  - ☞ *increment* the counter (add one to the internal number)
  - ☞ *read* or ”get” the current value of the counter
- This specification is useful *independent* of what you actually count!
- For example, a *stop watch* might be a special ”counter” with *auto-increment*

An Example Java Interface

- The *Counting* interface in Java may thus looks as follows:

```
// describe the name of the interface
public interface Counting
{
  // describes the increment contract
  public void increment();

  // describes the getValue contract
  public int getValue();
}
```

- Now that the interface has been defined, we can build a counting entity
- We can also use *other, foreign* counting entities
  - ☞ We only know that the methods `getValue()` and `increment()` are implemented
- However, we do *not* have to know *how* the methods are implemented.

An Example Java Interface

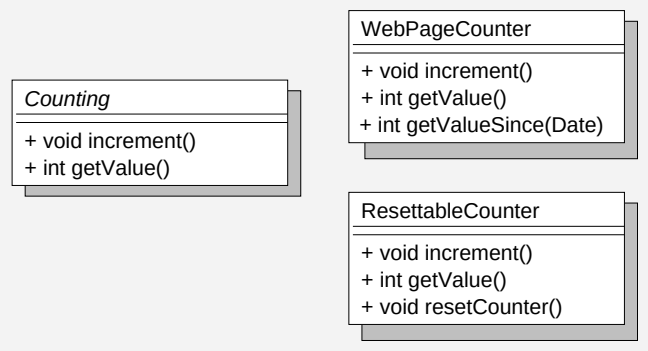

Figure 10: The *Counting* interface and two implementations

- However, we may want to know more about the entities
- Parts of the specification will be in the documentation:
  - ☞ *Counting* guarantees that the count is always positive
  - ☞ The *Resettable* counter may be reset using *resetCounter()*
  - ☞ The *WebPageCounter* can deliver the count *since* a given date

Method Signatures

- The *methods* we have already seen in the *StringTransformer* examples are *rules* for how to accomplish particular behaviors
- Interfaces focus on the *specification* of these rules, **not** on the instructions needed to achieve them
- Thus, an interface is a *collection of rule specifications*
- *All* objects implementing the interface must satisfy these specifications
  - ☞ *How* the implementors will do this is left to them!
- The formal name for a rule specification is *method signature*
- *Counting* specifies two methods: `increment` and `getValue`
- These methods must be provided by *all Counting* objects!

Method Signatures

- The method signatures consists of…
  - ☞ the *returned result* of the method,
  - ☞ the *name* of the method,
  - ☞ the entities the method *expects* as *parameters*.
  - ☞ Furthermore, the method may also specify possible ”error” (*exceptions*).
- The method signature does **not** require a body (implementation)
- The actual realization is left to the *implementing entities*
- *Users* only need to know the elements listed above

Method Signatures

- It is sensible to choose a *helpful name* for both the *interface* and its *method(s)*
- Parameters consist of *type* and *name* for each parameter
- We need also use helpful names for the *parameters*!
- If there is more than one parameter, they are separated with commas
- The *order* of the parameters is fixed - make a good decision in which order they have to be given!
- By convention, both *method* and *parameter names* start with a lower case letter
- In many cases, the method will return a *value*

  ☞ In this case, the *type* of the result must be placed before the method name
- The keyword `void` is used if *nothing* is returned

Method Signatures

- If we want to do something with the *returned value*, we have to *assign* it to a variable
- Let us assume that `myCounter` is of type *Counting*:

```
int counterValue = myCounter.getValue()
```

- `counterValue` now holds the value of the counter `myCounter` at the time of the method call

Abstract Method Declaration Syntax

❑ Here's the synopsis for declaring a *method*:

```
access returnType methodName(paramType1 paramName1,
                              ...,
                              paramTypeN paramNameN);
```

❑ access will typically be either *omitted* or public

❑ returnType can be any valid type

❑ paramType1, ..., paramTypeN are the types of the parameters

❑ The semicolon ; at the end states that this is a *specification*

❑ Such a method is called *abstract*; the abstract keyword *may* be prepended:

```
public abstract int getValue();
```

❑ *Interface* only specify signatures, the methods are *always* abstract

Open Questions

❑ The signatures leaves several questions unanswered

❑ For each parameter, we have to figure out...

- ☞ what the parameter is *intended to represent*,
- ☞ what *relationships* there might be between parameters,
- ☞ whether there are restrictions on the legal values,
- ☞ and whether the object represented by the parameter will be *modified*.

❑ For the return type, we have to find out...

- ☞ what the relationship of the returned object to the parameters is,
- ☞ what can be done with the result

Open Questions

- Furthermore, the following questions remain open, but may be relevant:
  - ☞ What are the *preconditions* for calling the method?
  - ☞ What can be *expected* of the result?
  - ☞ How *long* does the method take?
  - ☞ What other properties are used in the method?
- Not *all* of these questions are relevant to every method
  - ☞ The precise time needed for the `getValue()` method is probably not important - is should only be *reasonably quick*

Method Documentation Guidelines

- Why and when should users use the method?
- What does the method do?
- When is it appropriate to use the method, when inappropriate?
- Are there other methods that should be used instead, or in addition?
- Are there any *assumptions* made in the method?
- What is the meaning of each parameter?
- Which parameters are *modified* in the method?
- What *assumption* about parameter values are made?

Method Documentation Guidelines

- What does the return value of the method represent?
- What is the relation to the arguments or other entities?
- Are there *assumptions* about the return value?
- What else might be affected by the method execution?
- Are any *other entities* changed by the method?
  - ☞ this is called a *side effect*
- If there are additional assumptions, also include them!

Interface Declaration

- Place *each* Java interface in a separate file
- The file name must match the *interface name* and end with .java
  - ☞ The *Counting* interface is declared in the file `Counting.java`
- The declaration starts with **interface** followed by the interface *name*
- The *interface body* is a set of method signatures placed between {...}
- The *order* of the methods does *not* matter

```
// describe the name of the interface
public interface Counting
{
  // describes the increment contract
  public void increment();

  // describes the getValue contract
  public int getValue();
}
```

Method Footprints

- An interface may contain more than one method with the *same name*
- However, the *footprint* must be different
- The method footprint consists of
  - the *method name*
  - and the *orddered list* of *parameter types*
- The *return type* and the *parameter names* are *not* part of the footprint
- `void reset(int newValue)` has a different footprint from `void reset()`
- `void reset(int initial)` and `void reset(int newVal)` have the same footprint

Method Overloading

- An interface may have multiple methods of the same *name* but different *footprints*
- This is called *method overloading*
- Overloading is helpful if the interface has some *similar* methods that do (slightly) different things
- For example, Java offers two methods for converting reals to integers:

```
public int round(float f);
public long round(double d);
```

- Thus, a 32 bit `float` is rounded to a 32 bit `int`, and a 64 bit `double` is rounded to a 64 bit `long`

Interfaces Are Types

- Interfaces are also Java **types**
- *Every* interface name is also a **type name** in Java
- For example, when we need a *general counter*, we can state

```
Counting genericCounter;
```

- genericCounter is of type *Counting*
  - ☞ Thus, it will offer the *Counting* contract (interface)
- However, we do *not* know which *actual* type genericCounter may have - for example *ResetableCounter*

Interfaces Are Not Implementations

- An *interface* can be used as the *type* of an object
- *All* methods declared in the interface can be invoked on the object
- However, we do *not* know which *precise* implementer of the interface we have, and how it is implemented
- Keep in mind:

  Interfaces are contracts, not implementation details

- The next sections will cover how we can build implementation that fulfill the contracts

Interface Documentation Guidelines

- ❑ Why and when should users use the interface?
- ❑ What does the interface do?
- ❑ When is it appropriate to use the interface, when inappropriate?
- ❑ Are there other interfaces that should be used instead, or in addition?
- ❑ Are there any *assumptions* made in the interface?
- ❑ What services does the interface provide?
- ❑ How do you use the services?
  - ☞ This is typically covered in the *method documentation*
  - ☞ However, a short overview of all services should be included
- ❑ Which methods are appropriate in which context?

Interface Documentation Guidelines

- ❑ The documentation should make it easy to find the method(s) users want
- ❑ Users should also be able to determine if a new implementation fulfills the expectations
- ❑ Remember that the interface is mostly concerned with *what* it offers, not *how* this can be realized
- ❑ Thus, it will give a *contract* and *promises*, not *implementation details*

javadoc

❑ *JavaDOC* example!!! ☞ as separate slide set!

Expressions

❑ *Expressions* are used to transform *one* thing into an other

❑ We have already seen the *simplest* form of expression: *literals* and *names*

❑ Expressions also cover

- ☞ arithmetic including `+`, `-`, `*`, `/`, `(`, `)`
- ☞ logic including *and*, *or*, *not*,
- ☞ *methods* that return a value – such as `public String transform(String s)`

❑ After this chapter, we should. . .

- ☞ understand that *expressions* are Java code with a *type* and a *value*,
- ☞ be familiar with the *evaluation* of basic Java expressions,
- ☞ be able to understand complex expressions as combinations of simpler expressions,
- ☞ be able to evaluate simple and complex expressions.

Simple Expressions

- Expressions are the *simplest* piece of Java code
- As expressions are "*things*", they have both *value* and *type*
- When the code is executed, the expression is *evaluated*
- Evaluation yields the *value* of the given *type*
- Some of the expressions we have seen already include
  - `myName + " says "`
  - `"Obviously, "`

Literals

- The simplest Java expression is a *literal*
  - the value of a *literal* is interpreted *literally*
- Examples include 25, 2.5, "Hello"
- Literals can be *numbers*, *characters*, *Strings* or *boolean values*
- The *type* of a literal is what the literal looks like
  - 7 has type *integer*, "7" *String*, and *'7' character*
- The expression's type is the type of the value
  - For example, *integers* have type `int` , real numbers have type `double`

Names

- Names are *also* Java expressions
- However, the name must be *declared* in the context before it is used
- The *value* of the name is the value stored in the associated variable
- The *type* of the name is the type it was *declared with*
- In the following example code

```
int i = 42;
```

i has *value* 42 and *type* int

- ☞ if we evaluate i, we will get the int value 42.

Method Invocation

- The *services* that objects provide are called *methods*
- *Calling* a method is called *method invocation*
- *Method invocation* is the primary way to get one object to do something
- Java method invocations involve in this order
  - ☞ an expression of the *target object* whose service is requested,
  - ☞ a period ”.”
  - ☞ the *name* of the method to be invoked,
  - ☞ a pair of parentheses (), possibly including the *expressions* needed as parameters. The parentheses **must** be given, even if they contain nothing!
- One example invocation is `”testString”.toUpperCase()`
  - ☞ ”testString” is the *target object*, toUpperCase() the method to invoke
  - ☞ As toUpperCase() needs no parameters, the parentheses are empty.

Method Invocation

- Another example is the transform method:

```
String transform(String thePhrase)
```

- Here, a String object must be passed as an argument
- The *value* of the method invocation expression is its *return type*
  - ☞ This is the type given before the method name - here, it is String
- Some methods do *not have* a return value – their value is void
- One such method is the System.out.print(String) method
  - ☞ This method simply prints out the *expression* passed - it does not generate a result!

Method Invocation Evaluation

- To evaluate a method invocation, we can proceed as follows:
  1. evaluate the *object expression* of the invocation target object
  2. evaluate any argument expressions,
  3. evaluate the method invocation by asking the object to perform the method using the arguments
  4. The *value* of the expression is the value returned y the method invocation. The *type* of the invocation is the *declared* return type of the method.
- To make step 3 work, the object must know how to perform the method and return the result.
  - ☞ This is covered in the next chapter
- However, for the user, the exact process in step 3 is *of no concern*

Combining Expressions

- As we said before, expressions are again *things* - with *type* and *value*
- Therefore, we can *combine* expressions to build more complex expressions
- For example, `2 + 16 * 8` is a combined expression consisting of an *addition* and a *multiplication* expression
- If you invoke a method expecting an int, the following arguments are all legal:
  - 42 (literal value),
  - 6 * 9 (expression using literal values),
  - i (assuming i has been declared as int before the invocation),
  - i * 4 + 7 - 1 (again assuming i to be of type int
  - and many others.

Evaluation of Combined Expressions

- The basic rule for evaluation is simple:
  1. Evaluate all subexpression,
  2. combine the values of the subexpressions to the value of the expression
- Evaluating `"testString".toUpperCase()` thus works as follows:
  1. Determine the value of the literal "testString": "testString"
  2. Evaluate the method invocation: "TESTSTRING"
- The evaluation of the subexpressions may also contain further subexpressions
  - apply the same rule to the subexpressions
- We also have to evaluate the subexpressions in *the correct order*
  - 5 + 3 * 2 should return 11, not 16!

Assignments

- *Assignments* are also a form of expression
- The *left* side of the assignment operator = must be a *name* or *expression* that can refer to a *label* – not only a *value*
  - ☞ if this is the case, we call the left side an *assignable expression*
- The right-hand side can be any *expression*
- However, the *type* of the expression on the right and left side must match
- The *type* of the assignment is the type of the operand on the *left* side
- The *value* of the assignment is the value of the expression on the *right* side
- Thus, `int myInt = 42;` has type int and value 42.
- We will *not* use the value of assignments in expressions.

Other Expressions Using Objects: *Fields*

- *Fields* are data members of an object: think of NameDropper's name!
- Fields are accessed using the period notation, just as methods are - but without the parentheses
- The *value* is stored in the object and retrieved on access
- The *type* of a field access is the type of the declaration
- For example, Math.PI is a field of *type* double with value 3.14159...
- Field accesses can be combined with other expressions, if applicable
- As the field *stores* the data, you can use it on both the right *and* left side of an assignment
- myNameDropper.name = "Herb" changes the content of field name in a NameDropper
  - ☞ We will later learn how to prevent *unwanted* field changes

Instance Creation

- The new operator can be used to generate a new object
- new must be followed by a *class name*, for example NameDropper, and a *parameter list*
- Thus, the basic form is

new *ClassName*(*parameters*)

  - The words in *italics* have to be replaced by concrete values
- For example, a valid creation of a new NameDropper might be

```
new NameDropper("Herb");
```

- Typically, the result of new is assigned to a variable or field
- As with method invocations, the parameter list may be empty
- The *type* of new expression is the type of the object created
- The *value* of the new expression is the *new instance* of the requested class

Type Membership

- The last operator usable only with objects is instanceof
- instanceof tests, whether a given *object* has (or can have) a certain *type*
- The syntax for instanceof is as follows:

anObjectExpr **instanceof** *ObjectTypeName*

- *anObjectExpr* can be any expression that is an object – i.e., *not primitive*
- *ObjectTypeName* must be valid object type class
  - We can use the name of any class or interface
- The operator tests if the object on the left "matches" the type on the right and returns true or false.
  - Therefore,

```
"aString" instanceof String == true
new NameDropper() instanceof String == false
```

Operations on Primitive Types

- Most of the expressions in programs are *operations* on *primitive types*
- Foremost of the are the *arithmetic* and *logical* operations
- Each *operator* takes arguments of the specified types and returns a result of a particular type and value
- If x, y have type int, x + y will return an int
  - However, if x, y have type double, x + y will return a double
- There are three types of operators:
  - *unary* operators, which take only one argument,
  - *binary* operators taking two arguments – one to the left and the other to the right,
  - and one *ternary* operator with *three* arguments

Arithmetic Operators

op1 + op2	addition; also String concatenation
op1 - op2	subtraction; also used as *unary negation*
op1 * op2	multiplication,
op1 / op2	division,
op1 | op2	*bitwise* or
op1 & op2	*bitwise* and
op1 ^ op2	*bitwise* negation
op1 % op2	*remainder* of division (”modulo”)
op++	*postincrement*; equal to op = op + 1
op–	*postdecrement*; equal to op = op - 1
++op	*preincrement*; equal to op = op + 1
–op	*predecrement*; equal to op = op - 1

Table 4: Arithmetic operators for op, op1, op2 of type byte, short, int, long, float, double

- **Note:** `x=42; System.out.println(x++)` will **first** print the value of x (42), and **then** set x to 42+1 = 43!

Arithmetic Operators for Integral Types

op1	<<	op2	*left shift*, equal to op1 * 2^{op2}
op1	>>	op2	*right shift* with sign extension, equal to op1 / 2^{op2}
op1	>>>	op2	*right shift* with zero extension

Table 5: Arithmetic operators for op1, op2 of *integral* type (byte, short, int, long)

- byte and short are automatically converted to int before the operation
- If the types of both operands are the same (int, long, float, double), the value of the expression has the same type
  - **Thus, `5 / 2` has type int and returns 2**
  - For getting the "real" result, use `5.0 / 2` or `5 / 2.0`
- Otherwise, the *"wider"* type is chosen: long over int, double over float

Comparator Operators

op1	<	op2	test if op1 is less than op2
op1	<=	op2	test if op1 is less or equal to op2
op1	==	op2	test if op1 equals op2
op1	>=	op2	test if op1 is greater or equal to op2
op1	>	op2	test if op1 is greater than op2
op1	!=	op2	test if op1 is *not equal* to op2

Table 6: Comparator operators

- All comparator operations will return a boolean: either true or false
- Note the difference between *assignments* using = and *comparisons* using ==!
- *Two object labels* are only equal using == when they *refer to the same object*

Logical Expressions

op1 && op2	*logical* conjunction, ”and”
op1 || op2	*logical* disjunction, ”or”
! op	*logical* disjunction

Table 7: Logic operator expressions

- The operands op1, op2 must be expressions of type boolean
- The result type is also of type boolean
- && returns **true** only if *both* op1, op2 are **true**
- || returns **true** if *at least* one of op1, op2 is **true**
- !op returns **true** if op is **false**, and returns **false** if op is **true**

Parenthetical Expressions

- A *parenthetical expression* simply places a *parenthesis* around an expression
- The value of the expression is the value of the expression contained
- These expressions are *very* helpful when combining more complex expressions
  - ☞ They make expressions more readable
- They are also necessary for resolving *precedence*:

`2 + 3 * 5 == 2 + ( 3 * 5) == 17` but `( 2 + 3 ) * 5 == 5 * 5 == 25`

- **Hint:** Use parentheses whenever you are not *absolutely sure* how the expression will be evaluated!
- What happens here (for x=35, y=7)?

```
System.out.println(”The answer is ” + x + y);
```

Compound Assignments

leftSide	+=	expr;	same as	leftSide = leftSide + expr;
leftSide	-=	expr;	same as	leftSide = leftSide - expr;
leftSide	*=	expr;	same as	leftSide = leftSide * expr;
leftSide	/=	expr;	same as	leftSide = leftSide / expr;
leftSide	|=	expr;	same as	leftSide = leftSide | expr;
leftSide	&=	expr;	same as	leftSide = leftSide & expr;
leftSide	^=	expr;	same as	leftSide = leftSide ^ expr;
leftSide	%=	expr;	same as	leftSide = leftSide % expr;
leftSide	<<=	expr;	same as	leftSide = leftSide << expr;
leftSide	>>=	expr;	same as	leftSide = leftSide >> expr;
leftSide	>>>=	expr;	same as	leftSide = leftSide >>> expr;

Figure 11: Compound assignment operators

☞ leftSide must be an *assignable expression*

Ternary Expression Conditional

- There is only one *ternary* operator in Java : **( ) ? :**
- The form of the operator is

  (*logicalExpression***) ?** expression1 **:** expression2
- The *logicalExpression* is first evaluated
- Otherwise, the *value* is expression2
- expression1 and expression2 must have the same *return type*
- The return *type* of the operator is the type of the chosen *expression*
- If the result is **true** , the *value* of the operator is expression1
- The following example will print "even" for even x, else "odd":

```
System.out.println((x % 2 == 0) ? "even" : "odd");
```

- It is usually safer to be more verbose and *not* use **( ) ? :**

Type Conversion

- Sometimes, operations return a different type than we want
- For example, `Math.sqrt(9)` will return the *square root* of 9 as a **double** ☞ 3.0
- If we want to assign the value to an **int** , Java will complain!
- Java can automatically *widen* (”coerce”) primitive data types:
  - ☞ *byte* ☞ *short*,
  - ☞ *short* ☞ *int*,
  - ☞ *int* ☞ *long*,
  - ☞ *long* ☞ *float* (which possibly loses information!),
  - ☞ *float* ☞ *double*
- However, Java will not automatically *narrow* data types
- *Narrowing* might result in the loss of data
  - ☞ What would happen to the *square root* if we replace 9 by 8?

Explicit Type Conversion: ”Casting”

- *Casting* allows us to *force* Java to change the *viewed* type of an object
- The syntax is simple: **(** *targetTypeName* **)** *expression*
- Java now views the expression as having type targetTypeName

```
System.out.println(Math.sqrt(8.0)); ☞ 2.8284271247461903
System.out.println((int)Math.sqrt(8.0)); ☞ 2
```

- *Casting* may result in a **loss of information**
- `System.out.println((double)42);` ☞ 42.0 loses *no* information
- **Note:** the actual *type* of the expression is *not* changed by the casting!
- *Not all* casting operations are legal: the types must be *”conforming”*
  - ☞ Casting a **boolean** to **int** results in a *compile error*
  - ☞ Other operations may result in *runtime errors*

Order of Evaluation: *Precedence*

- For complex operations, we have to know the *order of evaluation*
  - ☞ Should `2 + 3 * 5` return `17` or `25` ?
- To resolve these issues, each operator has a *precedence*, shown in figure 8 on page 6.25
- The *higher* in the table an operator is, the *sooner* it will be evaluated
- Thus, since `*` is higher in the table than `+` , `2 + 3 * 5 == 2 + 15 == 17`
- The higher precedence operators claims its operators *first*

Order of Evaluation: *Precedence*

( ) as *parenthetical expression*
++, --, **unary** +, **unary** -,˜, !, explicit cast
*, /, %
binary +, **binary** -
<<, >>, >>>
<, <=, >, >=, **instanceof**
==, !=
&
^
|
&&
||
( ) ? :
= and all *compound assignments* such as +=

Table 8: Java operator precedence

Expressions Summary

- ❑ Every expressions has a *type* and a *value*
- ❑ Simple expressions include *literals* and *names*
- ❑ Operator expressions combine or modify simpler expressions
- ❑ The *autode- (++)* and *autoincrement(--)* operation **modify** the parameters

Operator	**Operation(s)**	***type* of result**
arithmetic	arithmetics	the *wider* of the operand types
assignment	value assignment	*type* of the value assigned
cast	type conversion	*type* of the cast operation
constructor call	generate new instance	*type* of the invoked constructor
field access	access data field	*type* of the declared field
logical	binary logic	always **boolean**
method call	method invocation	*return type* of the method

Table 9: Summary of Java operators

List of Figures

List of Tables

Listings

References

[1] Hornby, A. S. *Oxford Advanced Learner's Dictionary of Current English*, 3rd ed. Oxford University Press, 1987.

[2] Stein, L. A. *Interactive Programming In Java*. Morgan-Kaufmann Publishers, to appear.