Summary

Major Topics

1. static features (utility classes)
2. non-static features
3. mixing static and non-static features
4. aggregation and composition
5. inheritance
6. graphical user interfaces
7. recursion
8. data structures

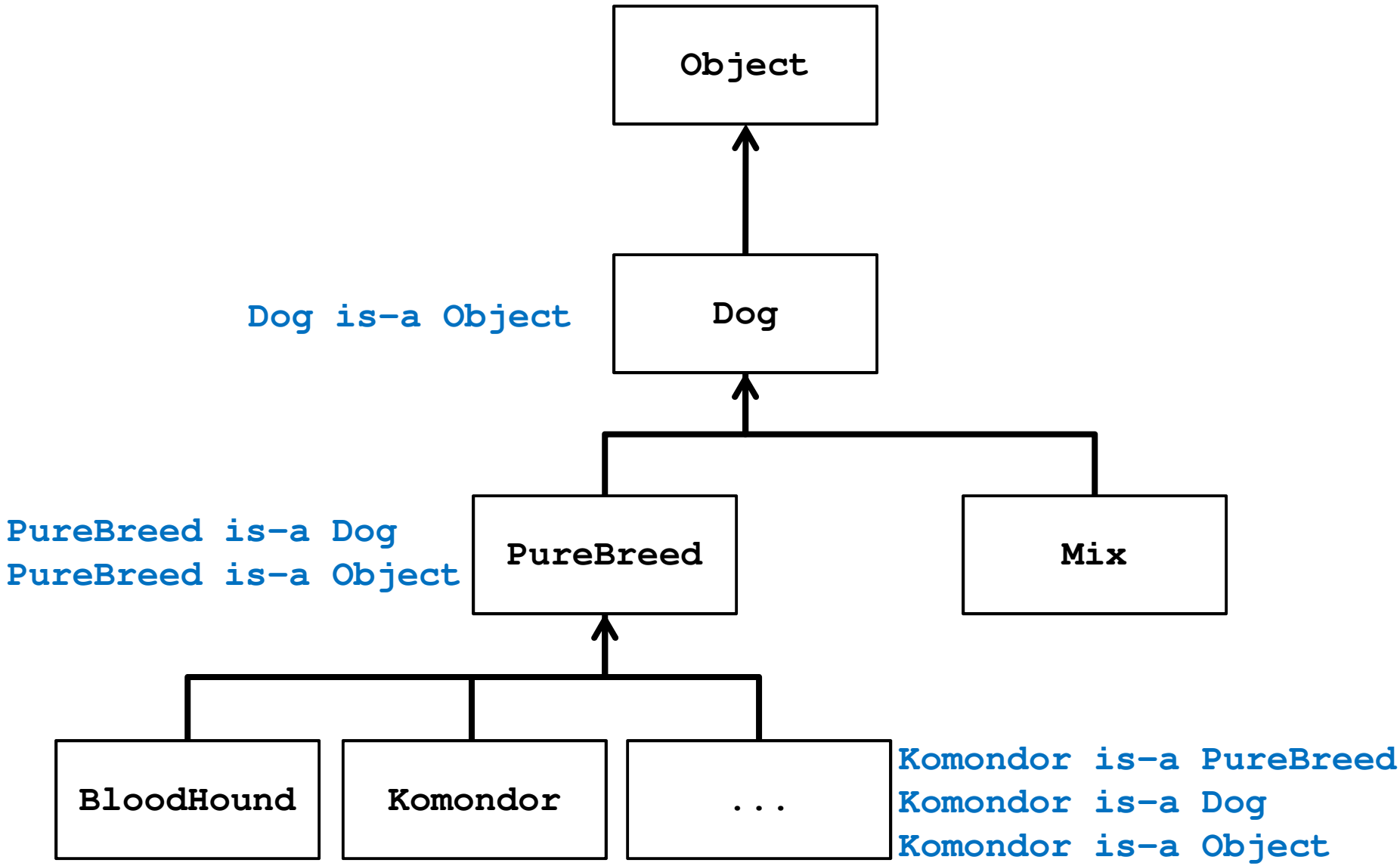
Inheritance

- means

is-a

or

is-substitutable-for

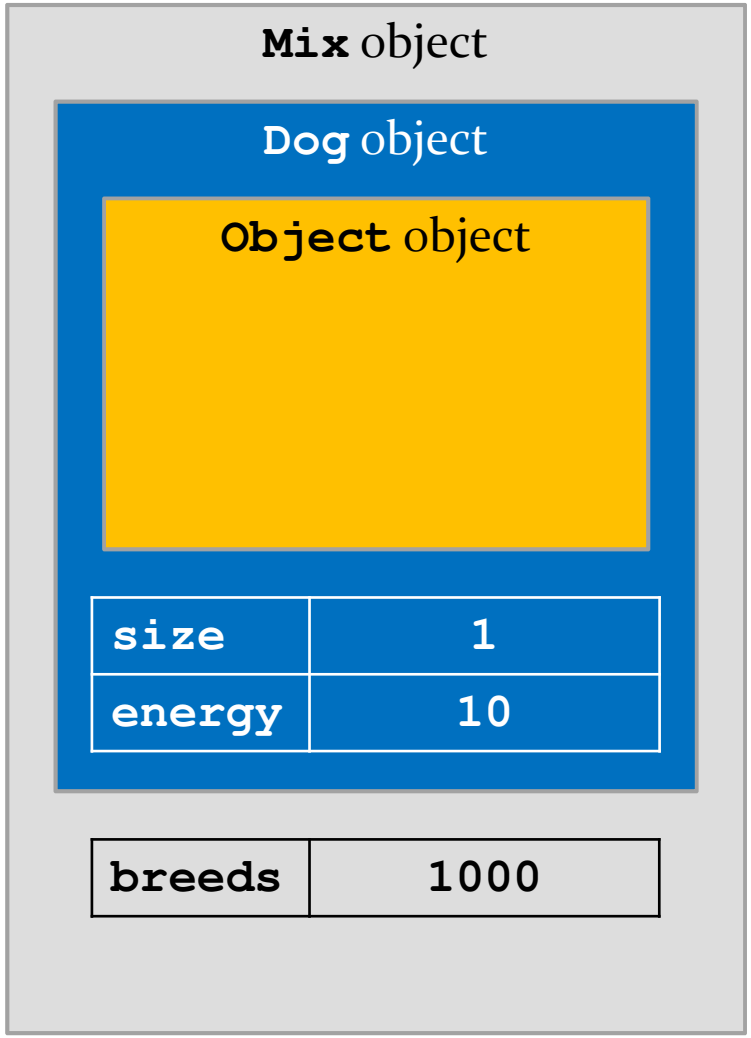

Object
Dog
Dog is-a Object
PureBreed
PureBreed is-a Dog
PureBreed is-a Object
Mix
BloodHound
Komondor
...
Komondor is-a PureBreed
Komondor is-a Dog
Komondor is-a Object

What is a Subclass?

- a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and attributes
- inheritance does more than copy the API of the superclass
  - the derived class contains a subobject of the parent class
  - the superclass subobject needs to be constructed (just like a regular object)
    - the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

```
Mix mutt = new Mix(1, 10);
```

1. **Mix** constructor starts running
- creates new **Dog** subobject by invoking the **Dog** constructor
    2. **Dog** constructor starts running
    - creates new **Object** subobject by (silently) invoking the **Object** constructor
        3. **Object** constructor runs
    - sets **size** and **energy**
- creates a new empty **ArrayList** and assigns it to **breeds**



Strength of a Precondition

- to strengthen a precondition means to make the precondition more restrictive

```
// Dog setEnergy
// 1. no precondition
// 2. 1 <= energy
// 3. 1 <= energy <= 10
public void setEnergy(int energy)
{ ... }
```

weakest precondition

strongest precondition

Preconditions on Overridden Methods

- a subclass can change a precondition on a method *but it must not strengthen the precondition*
  - a subclass that strengthens a precondition is saying that it cannot do everything its superclass can do

```
// Dog setEnergy
// assume non-final
// @pre. none

public
void setEnergy(int nrg)
{ // ... }
```

```
// Mix setEnergy
// bad : strengthen precond.
// @pre. 1 <= nrg <= 10

public
void setEnergy(int nrg)
{
  if (nrg < 1 || nrg > 10)
  { // throws exception }
  // ...
}
```

- client code written for **`Dog`**s now fails when given a **`Mix`**

```
// client code that sets a Dog's energy to zero
public void walk(Dog d)
{
  d.setEnergy(0);
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Strength of a Postcondition

- to strengthen a postcondition means to make the postcondition more restrictive

```
// Dog getSize
// 1. no postcondition
// 2. 1 <= this.size
// 3. 1 <= this.size <= 10
public int getSize()
{ ... }
```

weakest postcondition

strongest postcondition

Postconditions on Overridden Methods

- a subclass can change a postcondition on a method *but it must not weaken the postcondition*
  - a subclass that weakens a postcondition is saying that it cannot do everything its superclass can do

```
// Dog getSize
//
// @post. 1 <= size <= 10

public
int getSize()
{ // ... }
```

```
// Dogzilla getSize
// bad : weaken postcond.
// @post. 1 <= size

public
int getSize()
{ // ... }
```

Dogzilla: a made-up breed of dog that has no upper limit on its size

- client code written for **`Dog`**s can now fail when given a **`Dogzilla`**

```
// client code that assumes Dog size <= 10
public String sizeToString(Dog d)
{
  int sz = d.getSize();
  String result = "";
  if (sz < 4)         result = "small";
  else if (sz < 7)    result = "medium";
  else if (sz <= 10) result = "large";
  return result;
}
```

- remember: a subclass must be able to do everything its ancestor classes can do; otherwise, clients will be (unpleasantly) surprised

Exceptions and Inheritance

- a method that claims to throw an exception of type **`X`** is allowed to throw any exception type that is a subclass of **`X`**
  - this makes sense because exceptions are objects and subclass objects are substitutable for ancestor classes

```
// in Dog
public void someDogMethod() throws DogException
{
  // can throw a DogException, BadSizeException,
  //              NoFoodException, or BadDogException
}
```

- a method that overrides a superclass method that claims to throw an exception of type **`X`** must also throw an exception of type **`X`** or a subclass of **`X`**
  - remember: a subclass promises to do everything its superclass does; if the superclass method claims to throw an exception then the subclass must also

```
// in Mix
@Override
public void someDogMethod() throws DogException
{
  // ...
}
```

checked exception

Which are Legal?

- in Mix

```
@Override
public void someDogMethod() throws BadDogException
```

```
@Override
public void someDogMethod() throws Exception
```

```
@Override
public void someDogMethod()
```

```
@Override
public void someDogMethod()
      throws DogException, IllegalArgumentException
```

technically legal, but don't do this

Abstract Classes

- abstract classes appear when there are common attributes and methods that all subclasses share
- often, only the subclasses will have enough information to implement the methods
  - these methods are marked abstract in the parent class to indicate that subclasses are responsible for providing the implementation

Static Features and Inheritance

- non-private static attributes are inherited
  - but there is still only one copy of the attribute and it is in the parent class
- non-private static methods are inherited
  - but they cannot be overridden, they can only be hidden

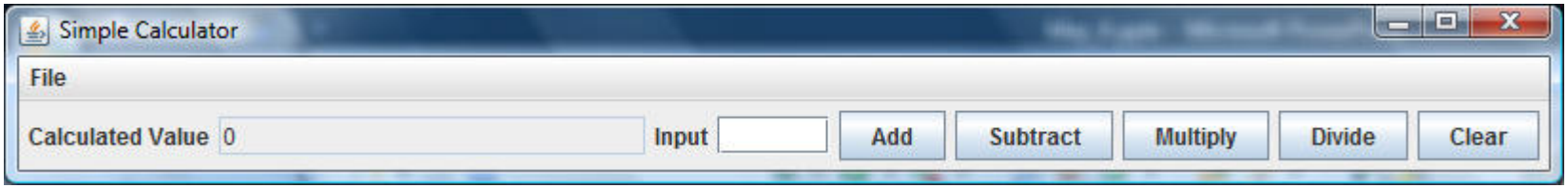
Interfaces

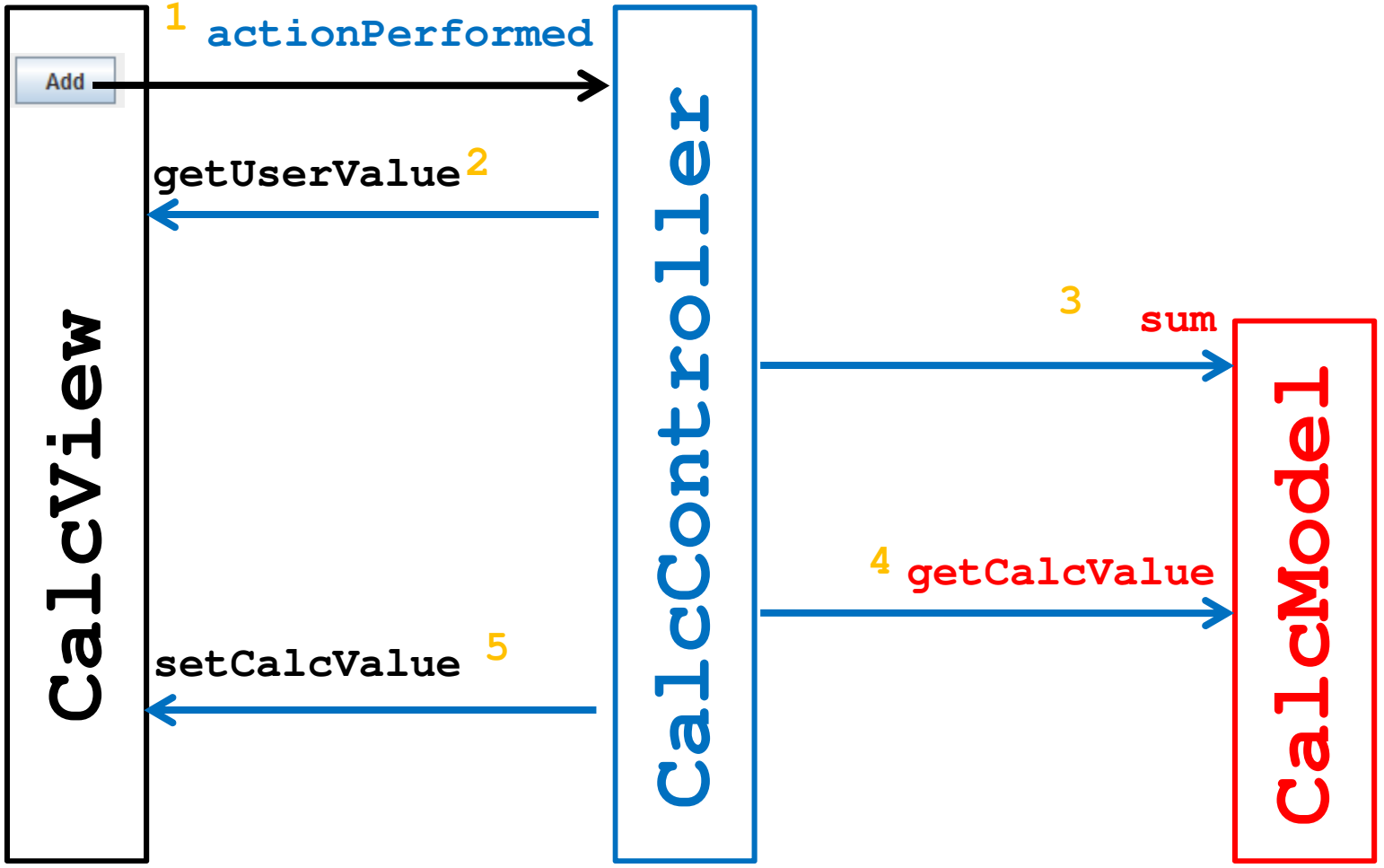
- in Java an *interface* is a reference type (similar to a class)
- an interface says what methods an object must have and what the methods are supposed to do
  - i.e., an interface is an API

- unlike inheritance, a class may implement as many interfaces as needed

Model-View-Controller

- model
  - represents state of the application and the rules that govern access to and updates of state
- view
  - presents the user with a sensory (visual, audio, haptic) representation of the model state
  - a user interface element (the user interface for simple applications)
- controller
  - processes and responds to events (such as user actions) from the view and translates them to model method calls


Simple Calculator
File
Calculated Value
0
Input
Add
Subtract
Multiply
Divide
Clear


CalcView
Add
1 actionPerformed
getUserValue 2
CalcController
3 sum
4 getCalcValue
setCalcValue 5
CalcModel

Recursion

- a method that calls itself is called a *recursive* method
- a recursive method solves a problem by repeatedly reducing the problem so that a base case can be reached

```
printIt("*", 5)
*printIt("*", 4)
**printIt("*", 3)
***printIt("*", 2)
****printIt("*", 1)
*****printIt("*", 0) base case
*****
```

Notice that the number of times the string is printed decreases after each recursive call to printIt

Notice that the base case is eventually reached.

Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then prove that each recursive case is correct

Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n - 1)` times. Then the recursive case prints the string `s` exactly `(n - 1)+1 = n` times; thus the recursive case is correct.

Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

Termination of printIt

1. **`printIt(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printIt(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printIt(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **`n`**.

Recurrence Relation

- analyzing the runtime of an algorithm often leads to a recurrence relation $T(n)$, e.g.,
  - $T(n) = 2T(n / 2) + O(n)$
  - $T(n) = T(n - 1) + T(n - 2)$

- solving the recurrence can sometimes be done by substitution

Solving the Recurrence Relation

$$
\begin{aligned}
T(n) \quad &\rightarrow \quad 2T(n/2) + O(n) \\
&\approx \quad 2T(n/2) + n \\
&= \quad 2[\ 2T(n/4) + n/2\ ] + n \\
&= \quad 4T(n/4) + 2n \\
&= \quad 4[\ 2T(n/8) + n/4\ ] + 2n \\
&= \quad 8T(n/8) + 3n \\
&= \quad 8[\ 2T(n/16) + n/8\ ] + 3n \\
&= \quad 16T(n/16) + 4n \\
&= \quad 2^k T(n/2^k) + kn
\end{aligned}
$$

$T(n)$ approaches…

Solving the Recurrence Relation

$$T(n) = 2^k T(n/2^k) + kn$$

- for a list of length 1 we know $T(1) = 1$
  - if we can substitute $T(1)$ into the right-hand side of $T(n)$ we might be able to solve the recurrence

$$n/2^k = 1 \Rightarrow 2^k = n \Rightarrow k = \log(n)$$

Data Structures

- recursive
  - linked list
  - binary tree
- stack
- queue

Previous written exam question

Suppose that you have a **`Stack`** class that has only the following features:

- the elements are of type **`int`**
- a default constructor that creates an empty stack
- a method **`isEmpty`** that returns true if the stack is empty
- **`push`** and **`pop`** methods

Describe how you would write a (static) method that makes a copy of a stack. A postcondition of your method must be that the state of the stack when the method finishes is the same as when the method started. Try to avoid using additional data structures (such as lists and arrays) if possible. Functional Java code is not required.

Previous written exam question

Suppose that you have a **`Queue`** class that has only the following features:

- the elements are of type **`int`**
- a default constructor that creates an empty queue
- a method **`size`** that returns the number of elements in the queue
- **`enqueue`** and **`dequeue`** methods

Describe how you would write a (static) method that makes a copy of a queue. A postcondition of your method must be that the state of the queue when the method finishes is the same as when the method started. Try to avoid using additional data structures (such as lists and arrays) if possible. Functional Java code is not required.