COMP 354: Introduction to Software Engineering

OO Fundamentals

Daniel Sinnig, PhD
d_sinnig@cs.concordia.ca

Department for Computer Science
and Software Engineering

Université Concordia University

04-June-2014

OO Fundamentals

- Objects vs Classes
- Interface vs. Implementation
- Static (compile-time) vs. Dynamic (run-time).
- Types
  - Type Hierarchy
- Runtime representation of OO programs.

Classes

- Hold data.
- Offer services to other objects and classes in its community through its methods.
- Are the only means of creating objects.

Objects

- Hold data.
- Offer services to other objects in its community through its methods.

A Class in a Class-based language …

- Defines four kinds of feature

	Object	**Class**
Data	*non-static field*	*static field*
Methods	*non-static method*	*static method*

Class: Interface vs. Implementation

- Interface (or type):
  - publicized services made available to others.

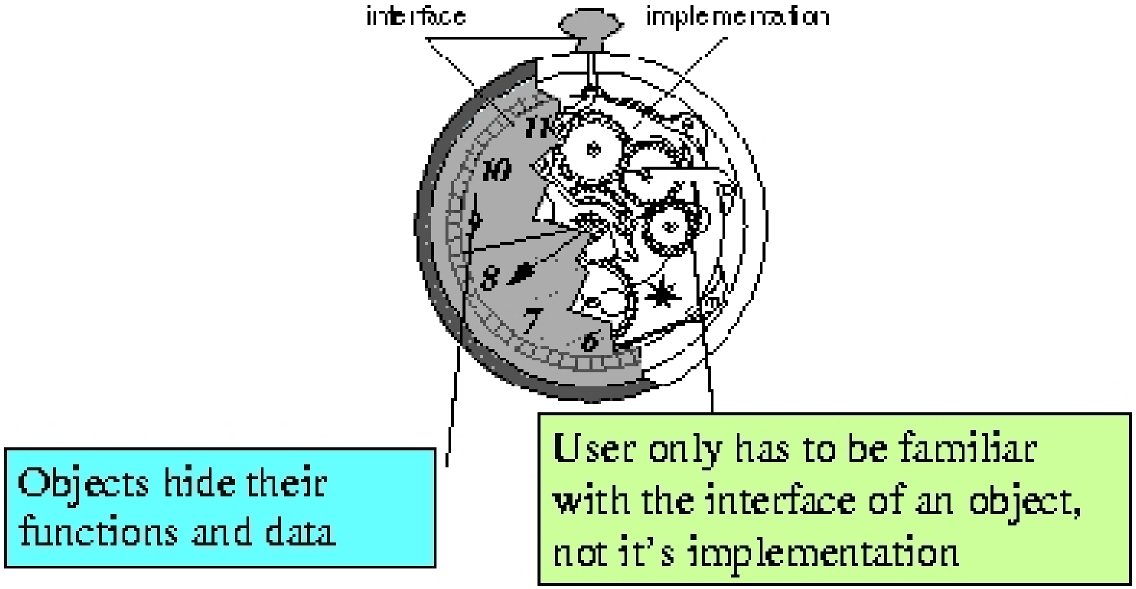
55
50
45
40
35
30
25
20
15
10
5
60
11
12
10
1
9
2
8
3
7
4
5
60
50
10
40
20
30

Class: Interface vs. Implementation

- Interface (or type):
  - publicized services made available to others.
- Implementation (generally hidden).

Class: Interface vs. Implementation

- Interface (or type):
  - publicized services made available to others.
- Implementation (generally hidden).

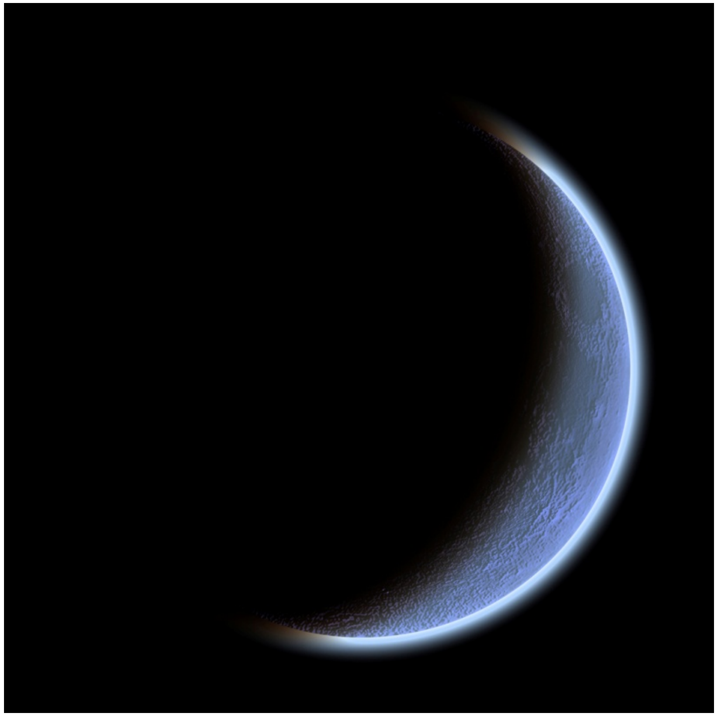


Objects hide their functions and data

User only has to be familiar with the interface of an object, not it's implementation

Two sides of an object

- Private features
  - Hidden
  - Help realize "information hiding"
  - Can be data or algorithm details
- Non-private features
  - Public
  - Package (or friend)

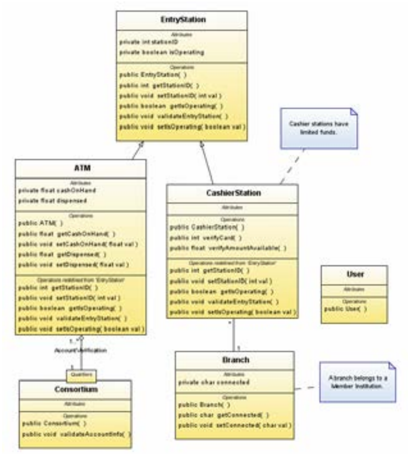


OO Programs are …

At Compile-time

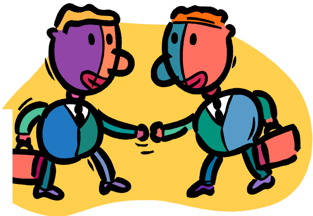
- Collection of *classes* … organized into *packages*.

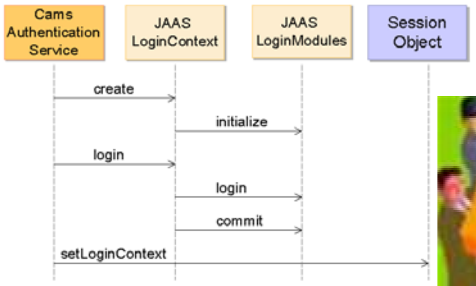
(static point of view)


EntryStation
Attributes
private int stationID
private boolean isOperating
Operations
public EntryStation( )
public int getStationID( )
public void setStationID(int val)
public boolean getIsOperating( )
public void validateEntryStation( )
public void setIsOperating(boolean val)
Cashier stations have limited funds.
ATM
Attributes
private float cashOnHand
private float dispensed
Operations
public ATM( )
public float getCashOnHand( )
public void setCashOnHand(float val)
public float getDispensed( )
public void setDispensed(float val)
Operations redefined from 'EntryStation'
public int getStationID( )
public void setStationID(int val)
public boolean getIsOperating( )
public void validateEntryStation( )
public void setIsOperating(boolean val)
CashierStation
Attributes
Operations
public CashierStation( )
public int verifyCard( )
public float verifyAmountAvailable( )
Operations redefined from 'EntryStation'
public int getStationID( )
public void setStationID(int val)
public boolean getIsOperating( )
public void validateEntryStation( )
public void setIsOperating(boolean val)
User
Attributes
Operations
public User( )
1..*
AccountVerification
1
Qualifiers
Consortium
Attributes
Operations
public Consortium( )
public void validateAccountInfo( )
*
1
Branch
Attributes
private char connected
Operations
public Branch( )
public char getConnected( )
public void setConnected(char val)
A branch belongs to a Member Institution.

At Runtime

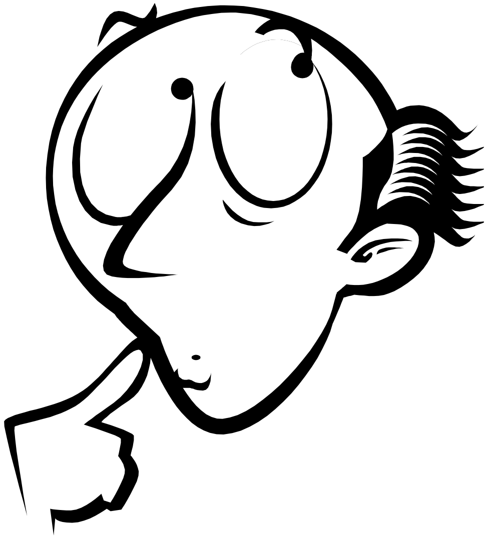
1. collaborating community of *objects* …
2. using *class features*:
   - data
   - methods





Question

- What are the different kinds of feature visibilities that can be used in Java?



OO Feature Visibility

- Private features
- Public features
  - Defines an interface
- Package features

Types

- In Java, C++, … every
  - Declaration
  - Expression

  has a type.

Two kinds of type (Java)

- Basic (primitive):
  - int,
  - long,
  - double, …
  - void.

- Reference:
  - Class
  - Interface
  - Array

Types vs. Classes

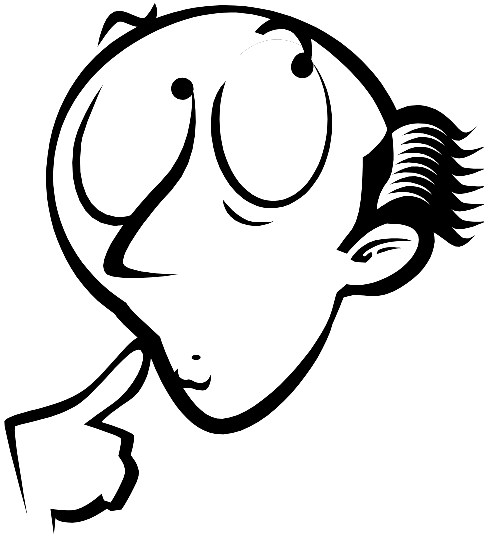
- A class (in either C++ or Java) defines
  - a type and
  - an implementation.

Types vs. Interfaces

- A Java interface defines
  - a type and
  - without an implementation.

Flash Question

- Can a type (without an implementation) be defined in any other way?



Type can be defined by means of

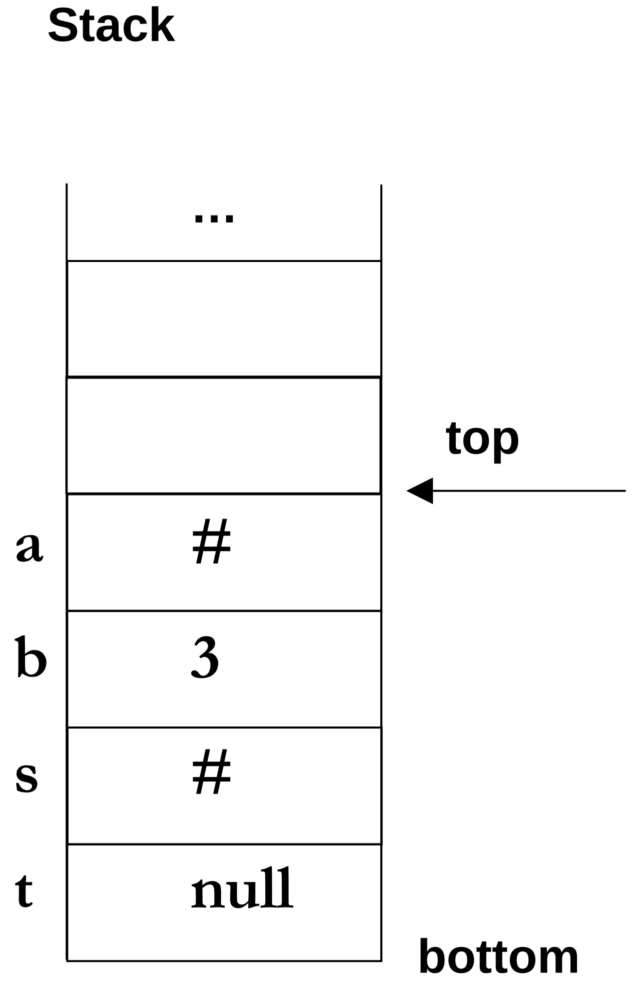
- Pure abstract class (C++, Java)
- Interface (Java)

Runtime Representation of OO Programs

- Stack and Heap.
- Runtime types of object.

Variables and the Stack

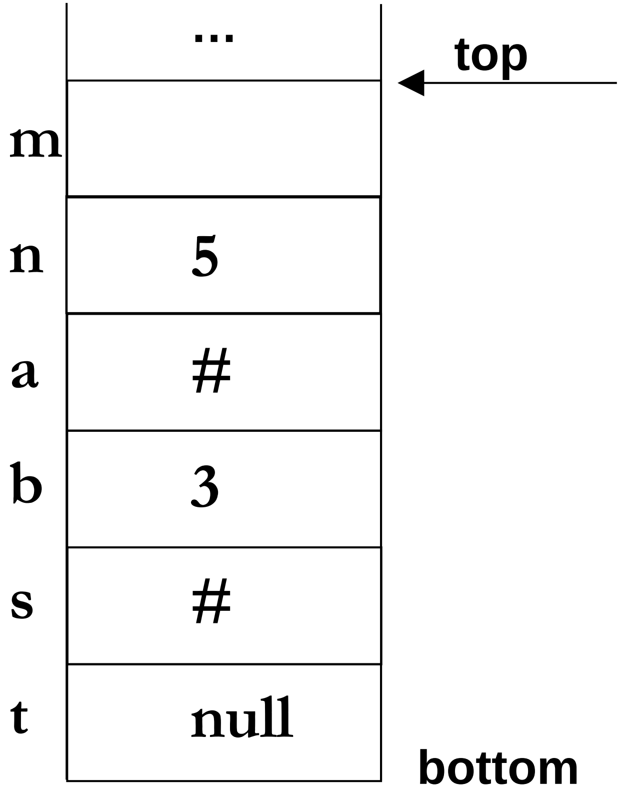
- Local variables (including method parameters) are allocated on the runtime *stack*.



Stack Frames and Method Calls

- Method call causes a *stack frame* to be pushed on the stack. Approximately as is shown: e.g. a call: m(5) where

```
void m(int n) {
  int x = …
  …
}
```



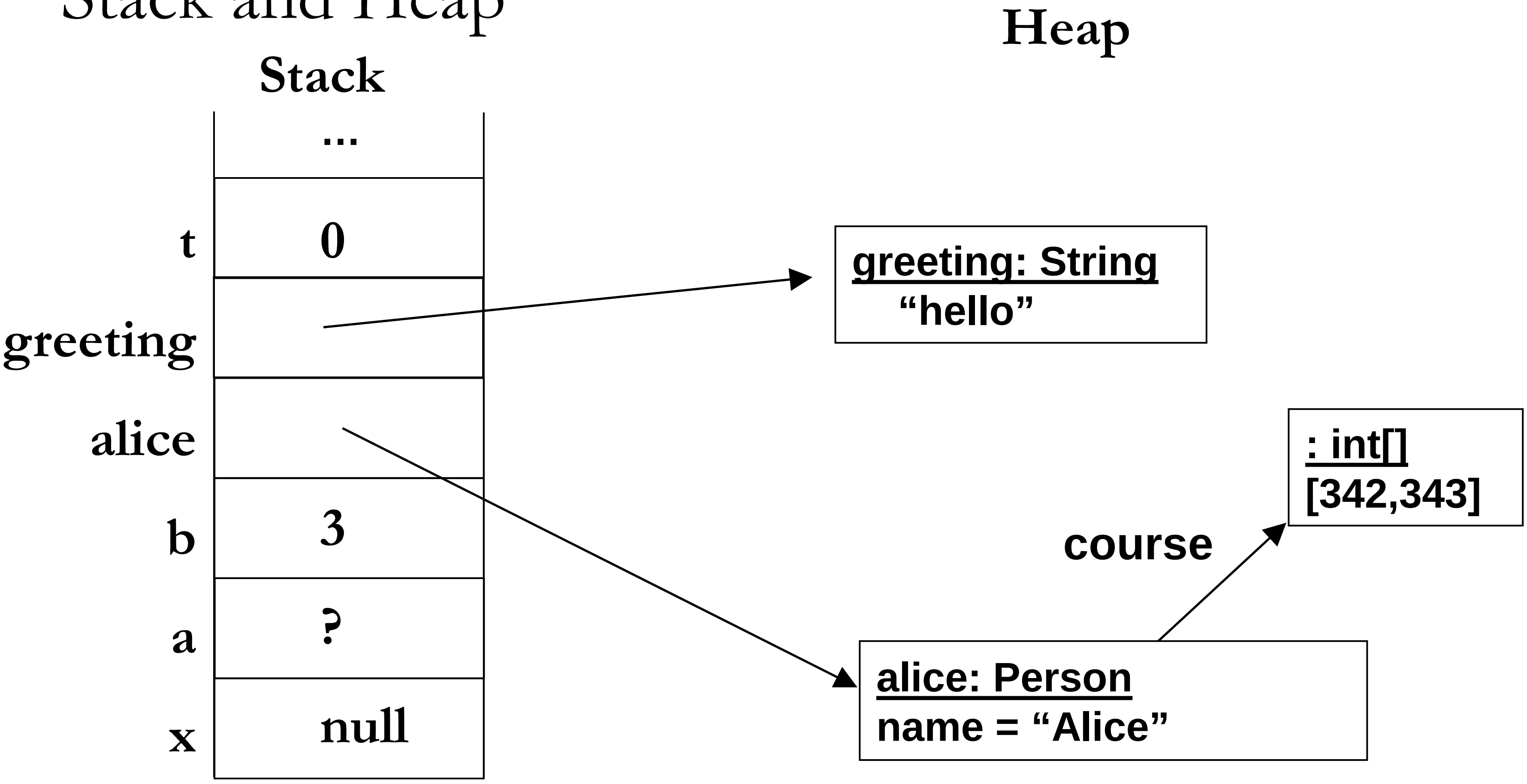
Variables and Types

- Every variable has a declared type that is either
  - Primitive type (int, char).
  - Reference type (int [], String, any class type).

Types, Values and the Stack & Heap

- Variables of
  - Primitive types contain values.
  - Reference types contain references to objects that are stored in the system *heap*.

Stack and Heap



Stack and Heap

- Stack: each stack cell can hold
  - The value of a primitive type.
  - The special value null.
  - A reference (pointer) to an object allocated in the heap.
  - (Objects cannot be held in the stack)
- Heap:
  - Only contains objects that were created via “new”.

Object and Variables: Run-time Initialization

- Local variables:
  - Are not implicitly initialized.
  - Must be explicitly initialized.
- Object fields always initialized by default to
  - Integral types (including char): `0`.
  - Floating point types: `0.0`.
  - Reference types: `null`.

Object Creation and Variable Declarations

- Basic declaration (*no* object creation):

```
Animal a;
```

- null initialized declaration (*no* object creation):

```
Animal a = null;
```

- Only using `new` will create (instantiate) objects

```
a = new Duck();
```

Exercise:

Illustrate the state of the stack and heap after the following local variable declarations have been processed.

```
int    i = 6;
int    j;
int    a[] = {1,2,3};
int[]  b = new int[2];
String s = “abc”;
String t = null;
```

Compile-time vs. Runtime type

- Any expression can have two types:
  - Compile-time (or static) type;
  - Runtime (or dynamic) type.
- For variables, the compile-time type is also called its declared type.
- E.g. consider a small Animal hierarchy …

Each Expression: Two Types, Example

```
Animal a = new Cat();
```

- `a` has two types:
  - Declared type: `Animal`.
  - Run-time type: `Cat`.

How can I “recover” the dynamic type?

```
Animal a = new Cat();
Cat c = a;               // ?
```

How can I “recover” the dynamic type?

```
Animal a = new Cat();
Cat c = (Cat)a;          // ?
```

Type Casting (in Statically Typed OO Languages)

- Purpose of Casting
  - Inform compiler of (assumed) subtype of an object.
  - Compiler can then perform better type checking.
- Type cast
  - Like an assertion, it may fail; e.g.

```
Object i = new Integer(0);
String s = (String) i;
// → ClassCastException
```

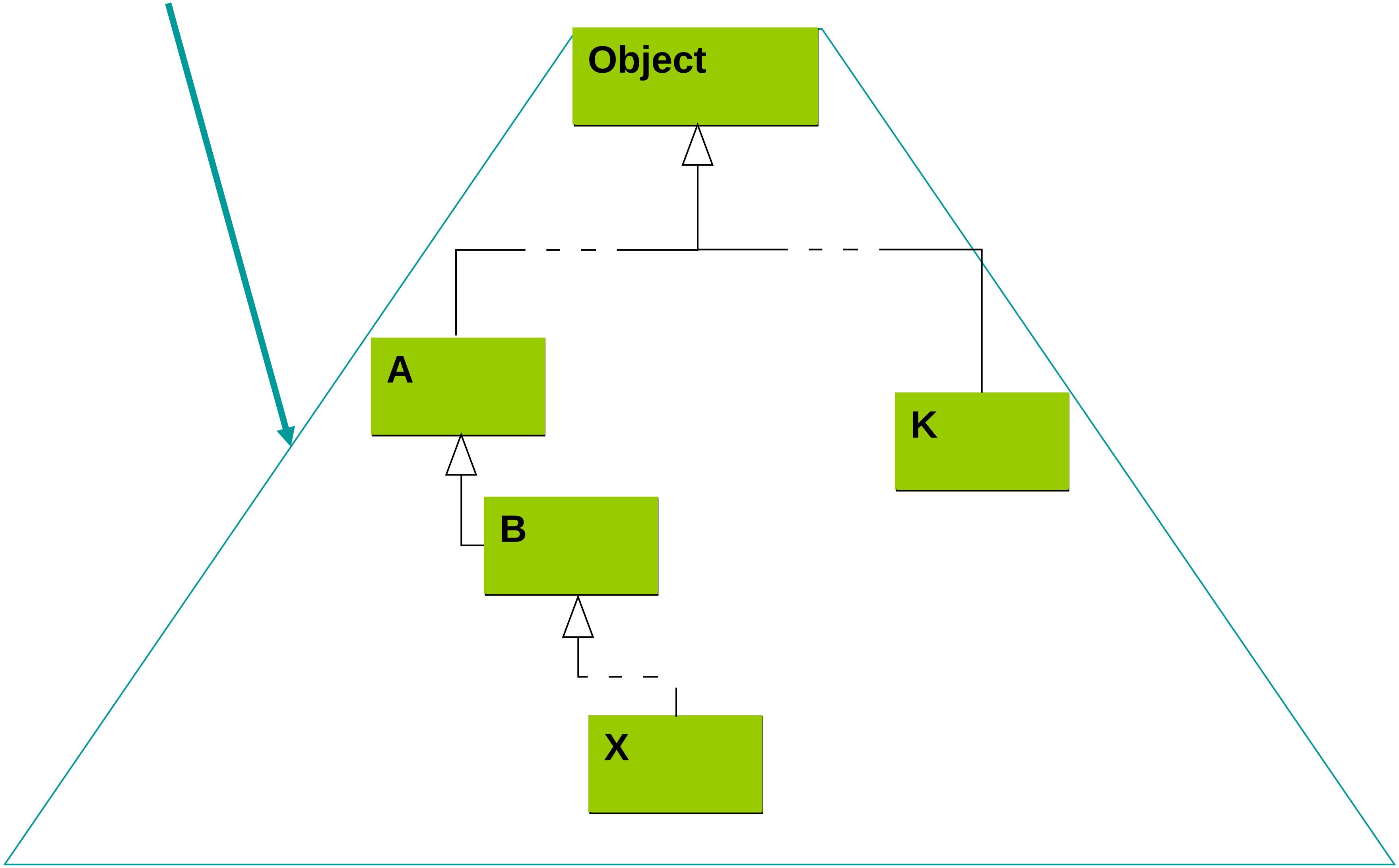
Type Hierarchy (Java)

- Every class is a subclass of `Object`.
- If S is a subclass of T then we can use an instance of S where ever a T is expected:

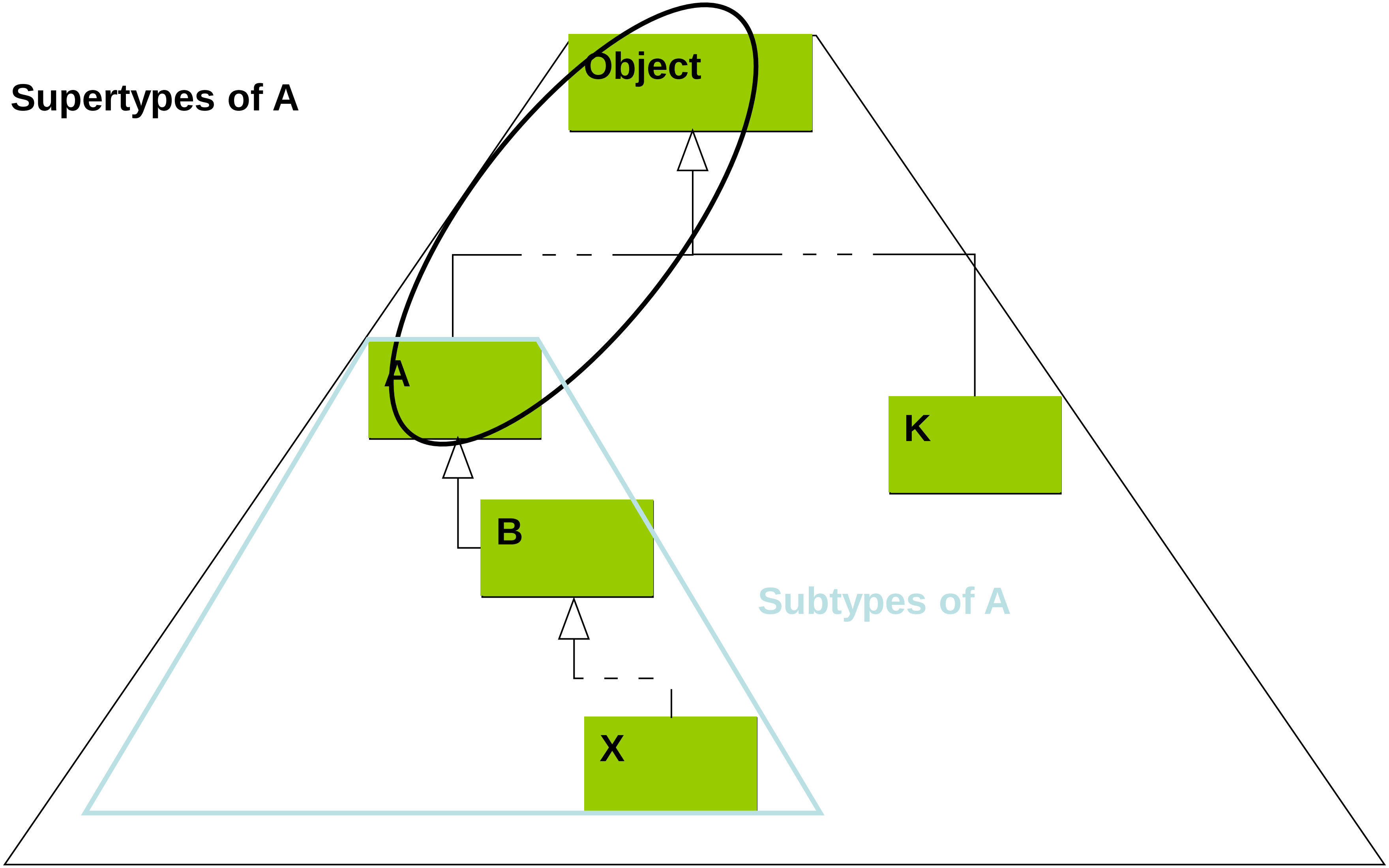
```
T t = new S();

Object o = new S();
// Do not do this (it is wasteful):
Object o = new String(“abc”);
```

Type Hierarchy: Object, the root of all



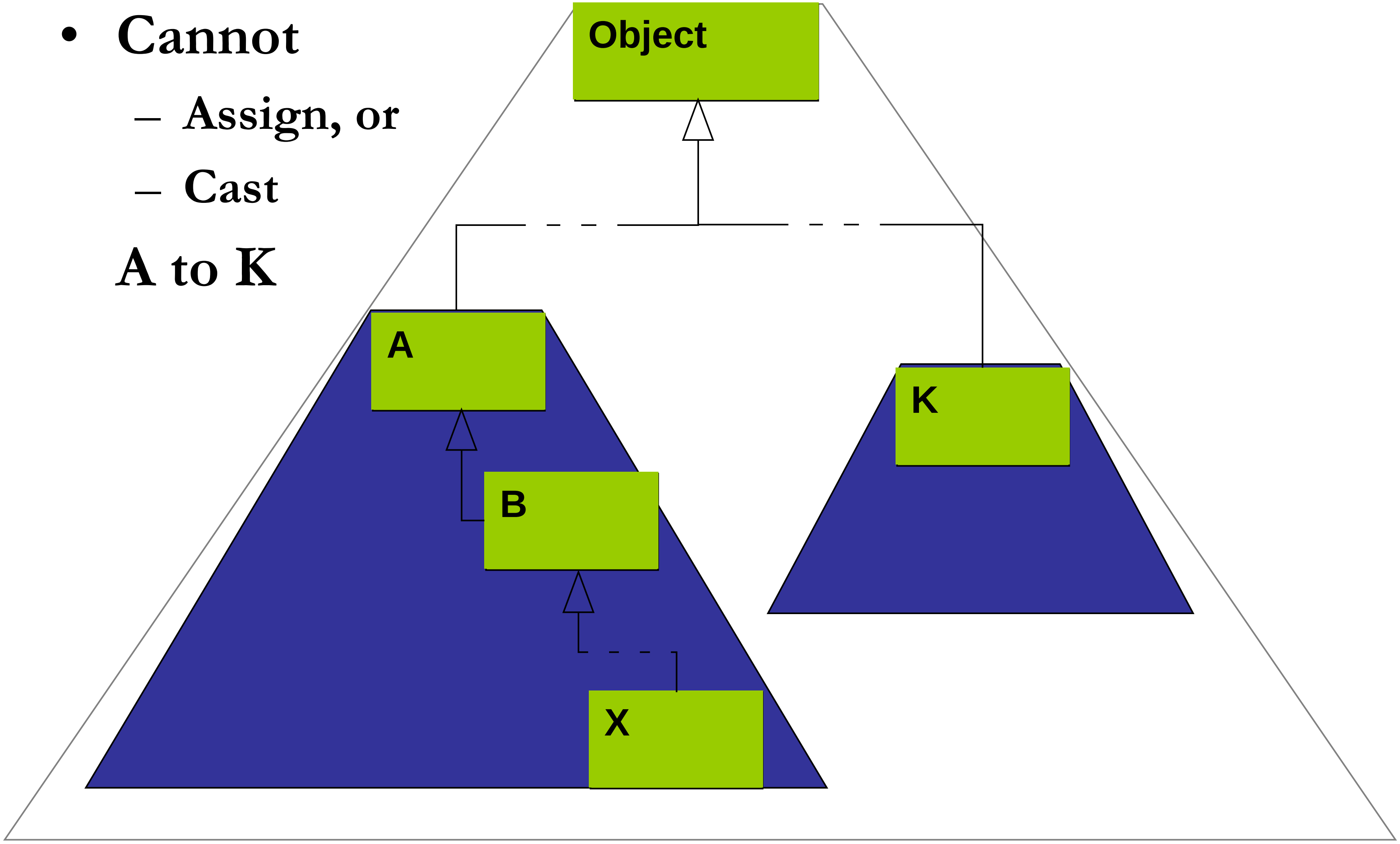
Type Hierarchy: Supertypes, Subtypes



Unrelated Sub-hierarchies are Not Compatible (for assignment, cast).

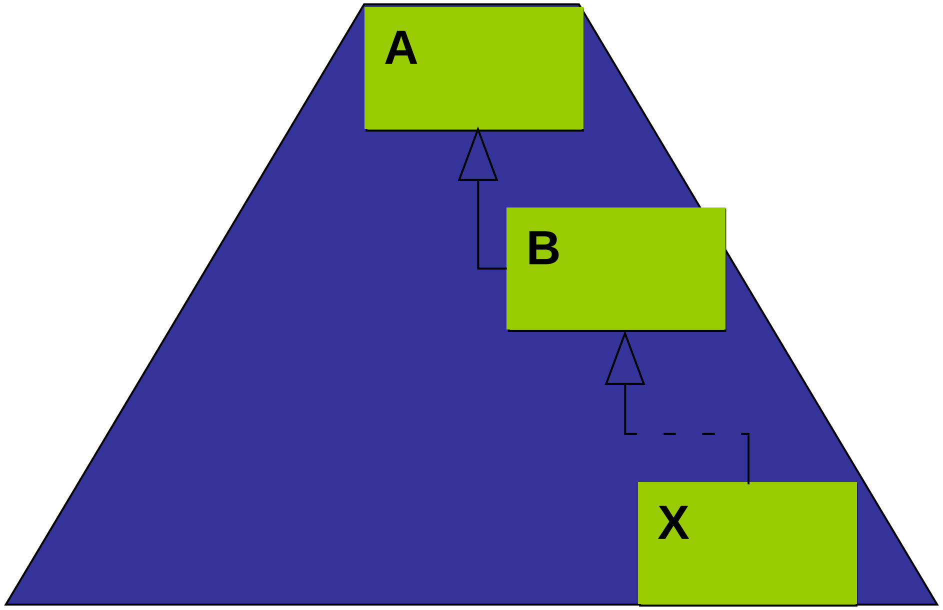
- **Cannot**
  - **Assign, or**
  - **Cast**

A to K



‘Declared Type’ Fixes Bounds

- **Declared type – e.g. A.**
- **Run-time type – can change at run-time …**
  - **… within bounds of** ***declared type subhierarchy*****.**



Type Checking: Assignment

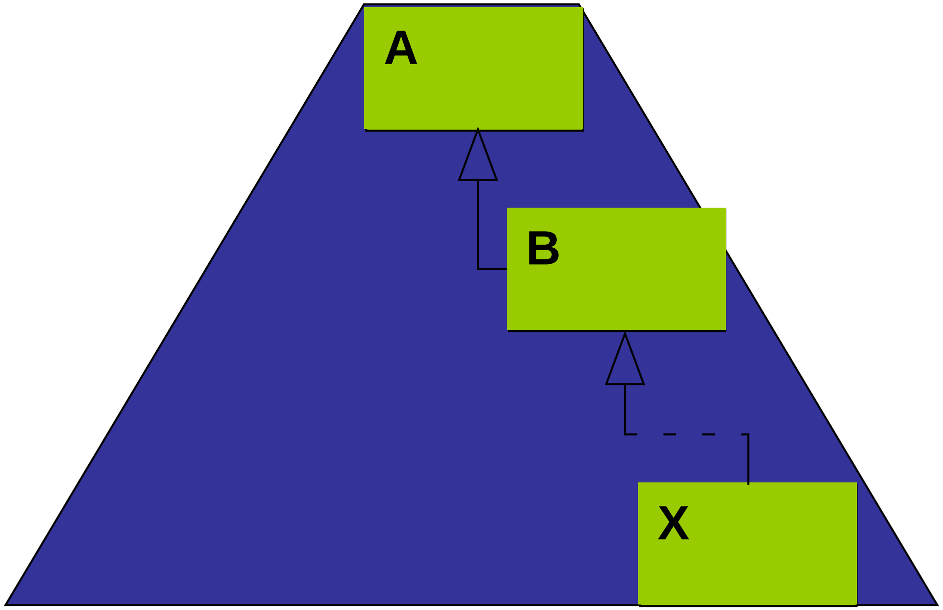
- **Always done relative to *declared* type.**

  A a = *some-expression-of-type-***X**

- **Legal? (or will cause a compile-time error?)**
- **Assignment is legal iff**

  X is

  - **A,**
  - **Subtype of A.**



Exercise:

Given the declarations below, what will be the value of b? Is this actually legal code; will an exception be thrown?

```
String s = “abc”;
Object o = “abc”;
boolean b = o.equals(s);
```

Static Methods (& Fields)

- Consider the call of a static method m():

  receiver.m()
- The declared type of the receiver expression determines which method m() gets called.
- Like a "global" procedure and/or variable.

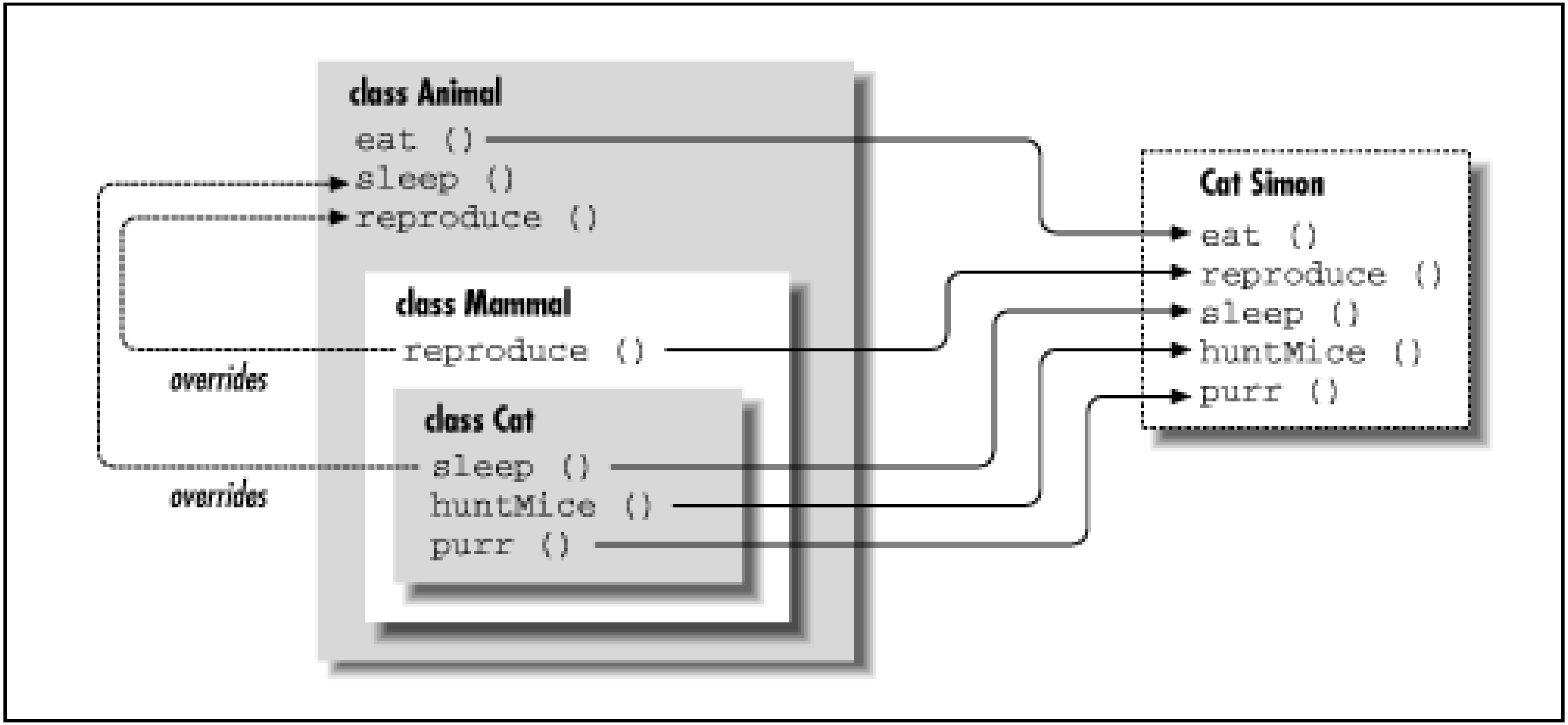
Non-Static Methods & Fields

- Let m() be a non-static method, then call:

  receiver.m()

- The method m() can be implemented in several classes (achieved by method overrides).
- Which implementation gets called depends on the run-time type of the receiver expression.
- This technique is called dynamic dispatching.

Small Animal Type Hierarchy



Subtype Polymorphism & Dynamic Dispatching

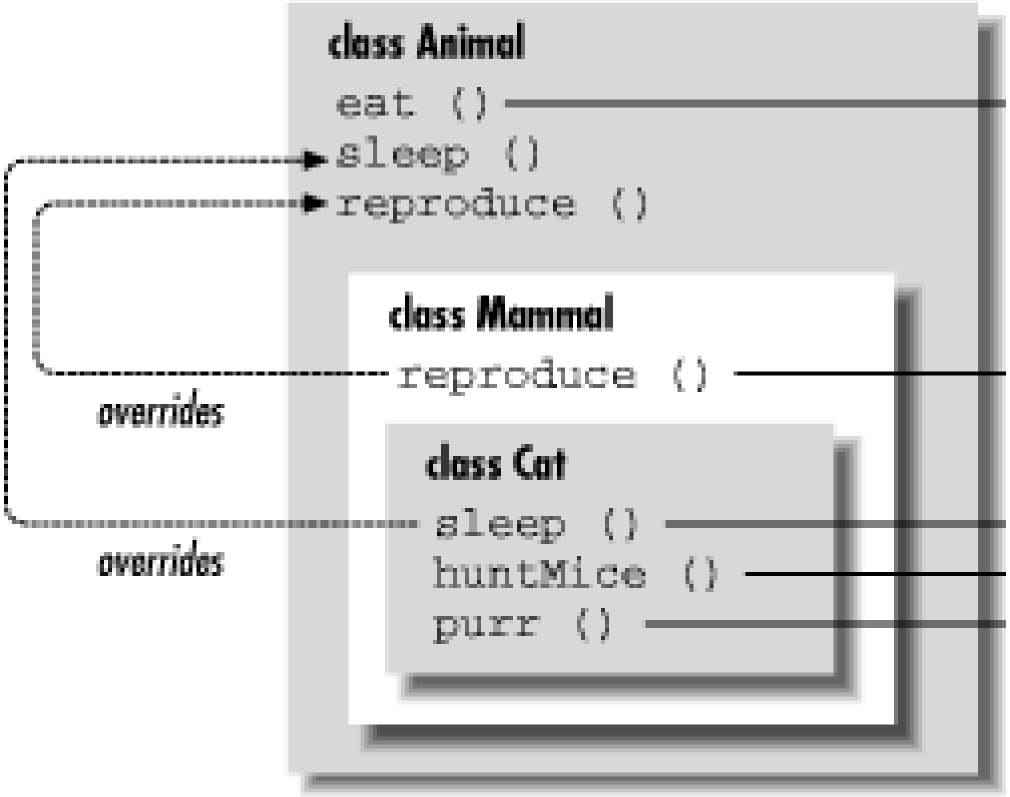
- Given

```
Animal a = new Cat();
a.sleep();
```

- Which sleep method gets called … ?

Subtype Polymorphism & Dynamic Dispatching

```
Animal a =
  new Cat();
a.sleep();
```



Polymorphism

- “poly” – many
- “morphism” – forms
- How does the meaning of this term apply to OO?
  - Run-time type of a given expression can vary.

- Different types: our concern
  - Subtype polymorphism.

Dynamic Dispatching

- Also called: dynamic method lookup.
- *Only* happens for non-static methods.
- NOT for any other method, field or constructor.

Dynamic Dispatching

Given *a*.m(), which implementation of m() will be called?

1. Determine the compile-time type of *a*. Let us called it A.
2. Look for a method m() with the appropriate signature in A or any supertype of A.
3. Not found? Then: Compiler-time error.
4. Found? And …

Dynamic Dispatching

Found, and it is a non-static method, then: …

- Determine the run-time type of *a*. (Call it X)
- Start looking up implementations of m():
  - Does X contain an implementation of m()?
    If yes, then use it.
  - Otherwise, move up the supertype hierarchy rooted at X one level at a time.

FYI – Vtables

- vtable lookup, involves a single table lookup.
  - Constructor initialized vtable entries appropriately.
- Contrast this with static methods where no lookup is needed.

Exercise:

Consider the given code (note that println represents System.out.println). What will the output be after running main()?

```
class P {
  static int rate = 1;
  static String m1() {
    return "P.m1";
  }
  static String m2() {
   return "P.m2";
  }
  String r1() {
   return "P.r1: " + rate;
  }
  String r2() {
   return "P.r2: " + rate;
  }
}
```

```
class C extends P {
  static int rate = 2;
  static String m1() {
    return "C.m1";
  }
  String r1() {
    return "C.r1: " +
rate;
  }
}
```

```
class Main {
  static public void main(String args[]) {
    P p = new C();
    println(p.m1());
    println(p.m2());
    println(p.r1());
    println(p.r2());
    println(p.rate);
    C c = new C();
    println(c.m1());
    println(c.m2());
    println(c.r1());
    println(c.r2());
    println(c.rate);
  }
}
```