Announcements

Lab next week:

- Peer code review for project 1. Attendance is **required**.
- Goals:
  - Help break you of bad programming habits!
  - Share good ideas.

Proj1B out Monday, maybe sooner. Study for exam.

- Proj1B is shorter than proj0 and proj1A.
- You'll use your Deque to solve a problem involving "palindromes", and will need to write your own tests.
  - Autograder for your code will be almost silent. We expect you to use your tests to fix your code.
  - Autograder will also test your tests (meta).

datastructur.es
CC BY NC SA

Announcements

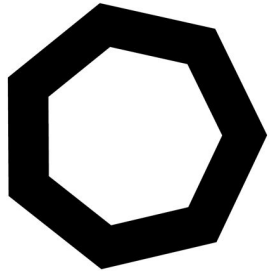
How to Study for an Exam:

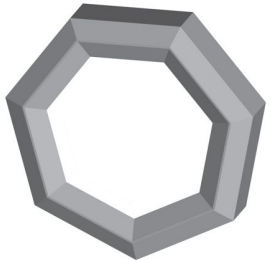
- Do lots of practice problems, but...
  - Carefully reflect on various techniques for solving them (best through discussion with peers).
  - Don’t look at solutions and try to “understand” them, or at least not until after you’ve already solved and discussed with others.
  - Help others work through problems. You learn a lot this way.

- Draw on the experience of your peers Link 1, Link 2

- See http://sp19.datastructur.es/materials/guides/study-guide.html for more.

datastructur.es
CC BY NC SA





CS61B, 2019

Lecture 8: Interface and Implementation Inheritance

- The Problem
- Hypernyms, Hyponyms, and Interface Inheritance
- Implementation Inheritance: Default Methods
- Implementation Inheritance: Extends

AList and SLList

After adding the insert methods from discussion 3, our AList and SLList classes have the following methods (exact same method signatures for both classes).

```
public class AList<Item>{
    public AList()
    public void insert(Item x, int position)
    public void addFirst(Item x)
    public void addLast(Item i)
    public Item getFirst()
    public Item getLast()
    public Item get(int i)
    public int  size()
    public Item removeLast()
}
```

```
public class SLList<Blorp>{
    public SLList()
    public SLList(Blorp x)
    public void insert(Blorp item, int position)
    public void addFirst(Blorp x)
    public void addLast(Blorp x)
    public Blorp getFirst()
    public Blorp getLast()
    public Blorp get(int i)
    public int   size()
    public Blorp removeLast()
}
```

Using ALists and SLLists: WordUtils.java

Suppose we're writing a library to manipulate lists of words. Might want to write a function that finds the longest word from a list of words:

```
public static String longest(SLList<String> list) {
    int maxDex = 0;
    for (int i = 0; i < list.size(); i += 1) {
        String longestString = list.get(maxDex);
        String thisString = list.get(i);
        if (thisString.length() > longestString.length()) {
            maxDex = i;
        }
    }

    return list.get(maxDex);
}
```

Observant viewers may note this code is very inefficient! Don't worry about it.

tastructur.es

Using ALists and SLLists: WordUtils.java

If we want longest to be able to handle ALists, what changes do we need to make?

```
public static String longest(SLList<String> list) {
    int maxDex = 0;
    for (int i = 0; i < list.size(); i += 1) {
        String longestString = list.get(maxDex);
        String thisString = list.get(i);
        if (thisString.length() > longestString.length()) {
            maxDex = i;
        }
    }

    return list.get(maxDex);
}
```

datastructur.es

Using ALists and SLLists: WordUtils.java

If we want longest to be able to handle ALists, what changes do we need to make?

```
public static String longest(AList<String> list) {
    int maxDex = 0;
    for (int i = 0; i < list.size(); i += 1) {
        String longestString = list.get(maxDex);
        String thisString = list.get(i);
        if (thisString.length() > longestString.length()) {
            maxDex = i;
        }
    }

    return list.get(maxDex);
}
```

datastructur.es

Method Overloading in Java

Java allows multiple methods with same name, but different parameters.

- This is called method **overloading**.

```
public static String longest(AList<String> list) {
    ...
}

public static String longest(SLList<String> list) {
    ...
}
```

datastructur.es

The Downsides

While overloading works, it is a bad idea in the case of `longest`. Why?

- Code is virtually identical. Aesthetically gross.
- Won't work for future lists. If we create a QList class, have to make a third method.
- Harder to **maintain**.
  - Example: Suppose you find a bug in one of the methods. You fix it in the SLList version, and forget to do it in the AList version.

datastructur.es

Hypernyms, Hyponyms, and Interface Inheritance

Hypernyms

In natural languages (English, Spanish, Chinese, Tagalog, etc.), we have a concept known as a “hypernym” to deal with this problem.

- Dog is a “hypernym” of poodle, malamute, yorkie, etc.

Washing your poodle:
1. Brush your poodle bef
2. Use lukewarm water.
3. Talk to your poodle in
4. Use poodle shampoo.
5. Rinse well. ...
6. Air-dry. ...
7. Reward your poodle.

Washing your **dog**:
1. Brush your **dog** before a bath. ...
2. Use lukewarm water. ...
3. Talk to your **dog** in a calm voice. ...
4. Use dog shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your **dog**.

nute:
ute before a bath. ...
ter. ...
nute in a calm voice. ...
mpoo. ...
mute.

Hypernym and Hyponym

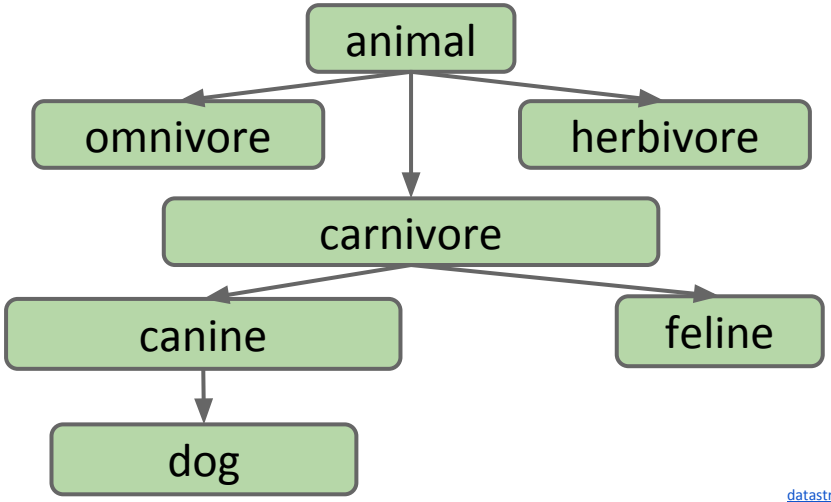
We use the word hyponym for the opposite type of relationship.

- “dog”: Hypernym of “poodle”, “malamute”, “dachshund”, etc.
- “poodle”: Hyponym of “dog”

Hypernyms and hyponyms comprise a hierarchy.

- A dog “is-a” canine.
- A canine “is-a” carnivore.
- A carnivore “is-an” animal.

(for fun: see the WordNet project)



datastructur.es

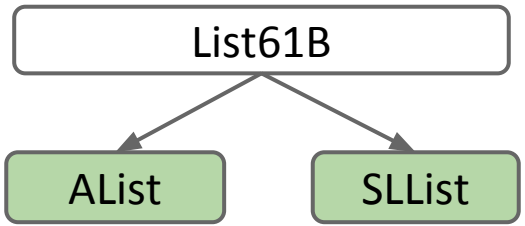
Simple Hyponymic Relationships in Java

SLLists and ALists are both clearly some kind of “list”.

- List is a hypernym of SLList and AList.

Expressing this in Java is a two-step process:

- Step 1: Define a reference type for our hypernym (List61B.java).
- Step 2: Specify that SLLists and ALists are hyponyms of that type.



datastructur.es

Step 1: Defining a List61B

We'll use the new keyword **interface** instead of **class** to define a List61B.

- Idea: Interface is a specification of what a List is able to do, not how to do it.

datastructur.es

Step 1: Defining a List61B

We'll use the new keyword **interface** instead of **class** to define a List61B.

- Idea: Interface is a specification of **what** a List is able to do, **not how** to do it.

```
public interface List61B<Item> {
    public void addFirst(Item x);
    public void addLast(Item y);
    public Item getFirst();
    public Item getLast();
    public Item removeLast();
    public Item get(int i);
    public void insert(Item x, int position);
    public int  size();
}
```

List61B

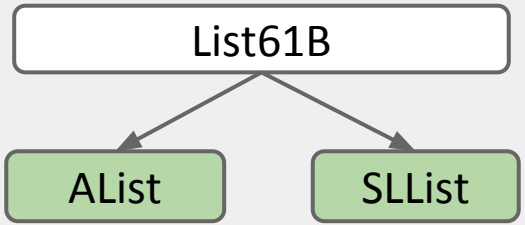
datastructur.es

Step 2: Implementing the List61B Interface

We'll now:

- Use the new **implements** keyword to tell the Java compiler that SLList and AList are hyponyms of List61B.

```
public class AList<Item> implements List61B<Item>{
    ...
    public void addLast(Item x) {
        ...
```



datastructur.es

Adjusting WordUtils.java

We can now adjust our longest method to work on either kind of list:

```
public static String longest(List61B<String> list) {
    int maxDex = 0;
    for (int i = 0; i < list.size(); i += 1) {
        String longestString = list.get(maxDex);
        String thisString = list.get(i);
        if (thisString.length() > longestString.length()) {
            maxDex = i;
        }
    }

    return list.get(maxDex);
}
```

```
AList<String> a = new AList<>();
a.addLast("egg");
a.addLast("boyz");
longest(a);
```

datastructur.es

Overriding vs. Overloading

Method Overriding

If a “subclass” has a method with the exact same signature as in the “superclass”, we say the subclass **overrides** the method.

```
public interface List61B<Item> {
   public void addLast(Item y);
   ...
```

```
public class AList<Item> implements List61B<Item>{
   ...
   public void addLast(Item x) {
      ...
```

AList overrides addLast(Item)

datastructur.es

Method Overriding vs. Overloading

If a “subclass” has a method with the exact same signature as in the “superclass”, we say the subclass **overrides** the method.

- `Animal`’s subclass `Pig` overrides the `makeNoise()` method.
- Methods with the same name but different signatures are **overloaded**.

```
public interface Animal {
   public void makeNoise();
}
```

```
public class Pig implements Animal {
  public void makeNoise() {
    System.out.print(“oink”);
   }
}
```

`Pig` overrides `makeNoise()`

```
public class Dog implements Animal {
   public void makeNoise(Dog x) {
      ...
```

`makeNoise` is overloaded

```
public class Math {
   public int    abs(int a)
   public double abs(double a)
```

`abs` is overloaded

datastructur.es

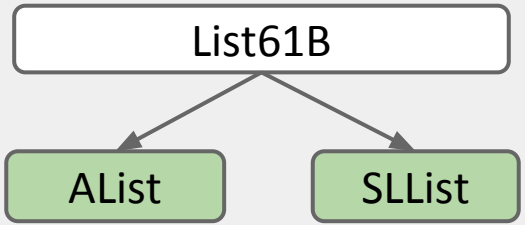
Optional Step 2B: Adding the @Override Annotation

In 61b, we'll always mark every overriding method with the **@Override** annotation.

- Example: Mark AList.java's overriding methods with **@Override**.
- The only effect of this tag is that the code won't compile if it is not actually an overriding method.

```
public class AList<Item> implements List61B<Item>{
    ...

    @Override
    public void addLast(Item x) {
        ...
```



datastructur.es

Method Overriding

If a subclass has a method with the exact same signature as in the superclass, we say the subclass **overrides** the method.

- Even if you don't write @Override, subclass still overrides the method.
- @Override is just an optional reminder that you're overriding.

Why use @Override?

- Main reason: Protects against typos.
  - If you say @Override, but it the method isn't actually overriding anything, you'll get a compile error.
  - e.g. `public void addLats(Item x)`
- Reminds programmer that method definition came from somewhere higher up in the inheritance hierarchy.

datastructur.es

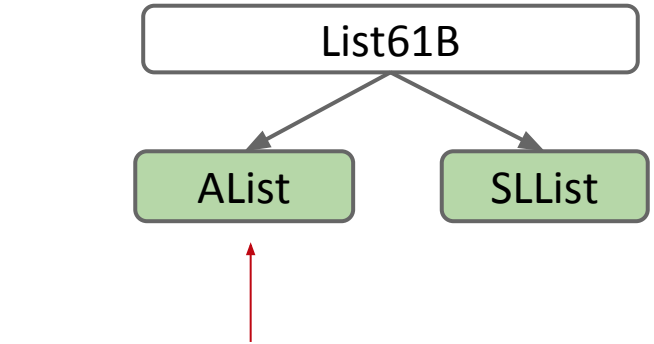
Interface Inheritance

Interface Inheritance

Specifying the capabilities of a subclass using the **implements** keyword is known as **interface inheritance**.

- Interface: The list of all method signatures.
- Inheritance: The subclass "inherits" the interface from a superclass.
- Specifies what the subclass can do, but not how.
- Subclasses must override all of these methods!
  - Will fail to compile otherwise.

```
public interface List61B<Item> {
    public void addFirst(Item x);
    ...
    public void proo();
}
```

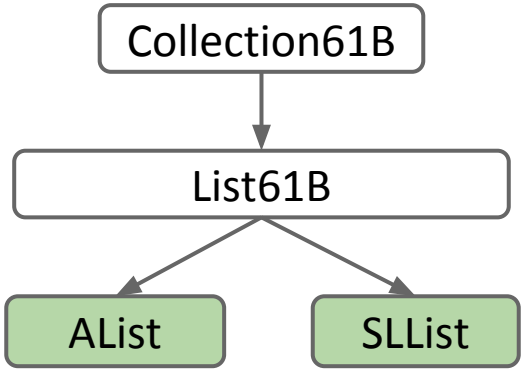


datastructur.es

Interface Inheritance

Specifying the capabilities of a subclass using the **implements** keyword is known as **interface inheritance**.

- Interface: The list of all method signatures.
- Inheritance: The subclass "inherits" the interface.
- Specifies what the subclass can do, but not how.
- Subclasses must override all of these methods!
- Such relationships can be multi-generational.
  - Figure: Interfaces in white, classes in green.
  - We'll talk about this in a later lecture.



Interface inheritance is a powerful tool for generalizing code.

- `WordUtils.longest` works on SLLists, ALists, and even lists that have not yet been invented!

datastructur.es

Copying the Bits

Two seemingly contradictory facts:

- #1: When you set x = y or pass a parameter, you're just copying the bits.
- #2: A memory box can only hold 64 bit addresses for the appropriate type.
  - e.g. **`String x`** can never hold the 64 bit address of a **Dog**.

```
public static String longest(List61B<String> list) {
    int maxDex = 0;
   for (int i = 0; i < list.size(); i += 1)
   ...
```

```
public static void main(String[] args) {
    AList<String> a1 = new AList<String>();
    a1.addLast("horse");
    WordUtils.longest(a1);
}
```

How can we copy the bits in **a1** to **list**?

Copying the Bits

Answer: If X is a superclass of Y, then memory boxes for X may contain Y.

- An AList is-a List.
- Therefore List variables can hold ALList addresses.

```
public static String longest(List61B<String> list) {
    int maxDex = 0;
   for (int i = 0; i < list.size(); i += 1)
    ...
```

```
public static void main(String[] args) {
    AList<String> a1 = new AList<String>();
    a1.addLast("horse");
    WordUtils.longest(a1);
}
```

How can we copy the bits in **a1** to **list**?

Question: http://yellkey.com/period

Will the code below compile? If so, what happens when it runs?

a. Will not compile.
b. Will compile, but will cause an error at runtime on the **new** line.
c. When it runs, an `SLList` is created and its address is stored in the `someList` variable, but it crashes on `someList.addFirst()` since the `List` class doesn't implement `addFirst`.
d. When it runs, an `SLList` is created and its address is stored in the `someList` variable. Then the string "elk" is inserted into the `SLList` referred to by `addFirst`.

```
public static void main(String[] args) {
   List61B<String> someList = new SLList<String>();
   someList.addFirst("elk");
}
```

datastructur.es

Question

Will the code below compile? If so, what happens when it runs?

a. Will not compile.
b. Will compile, but will cause an error at runtime on the **new** line.
c. When it runs, an `SLList` is created and its address is stored in the `someList` variable, but it crashes on `someList.addFirst()` since the `List` class doesn't implement `addFirst`.
d. When it runs, an `SLList` is created and its address is stored in the `someList` variable. Then the string "elk" is inserted into the `SLList` referred to by `addFirst`.

```
public static void main(String[] args) {
   List61B<String> someList = new SLList<String>();
   someList.addFirst("elk");
}
```

datastructur.es

Implementation Inheritance: Default Methods

Implementation Inheritance

Interface inheritance:

- Subclass inherits signatures, but NOT implementation.

For better or worse, Java also allows **implementation inheritance**.

- Subclasses can inherit signatures AND implementation.

Use the **default** keyword to specify a method that subclasses should inherit from an **interface.**

- Example: Let's add a default print() method to List61B.java

datastructur.es

Default Method Example: print()

```
public interface List61B<Item> {
   public void addFirst(Item x);
   public void addLast(Item y);
   public Item getFirst();
   public Item getLast();
   public Item removeLast();
   public Item get(int i);
   public void insert(Item x, int position);
   public int size();
    default public void print() {
       for (int i = 0; i < size(); i += 1) {
          System.out.print(get(i) + " ");
       }
       System.out.println();
    }
}
```

itastructur.es

Question: yellkey.com/computer

Is the print() method efficient?

a. Inefficient for AList and SLList
b. Efficient for AList, inefficient for SLList
c. Inefficient for AList, efficient for SLList
d. Efficient for both AList and SLList

```
public interface List61B<Item> {
    ...
    default public void print() {
        for (int i = 0; i < size(); i += 1) {
            System.out.print(get(i) + " ");
        }
        System.out.println();
    }
}
```

itastructur.es

Question

Is the print() method efficient?

a. Inefficient for AList and SLList
b. Efficient for AList, inefficient for SLList
c. Inefficient for AList, efficient for SLList
d. Efficient for both AList and SLList

```
public interface List61B<Item> {
    ...
    default public void print() {
        for (int i = 0; i < size(); i += 1) {
            System.out.print(get(i) + " ");
        }
        System.out.println();
    }
}
```

get has to seek all the way to the given item for SLists.

itastructur.es

Overriding Default Methods

If you don't like a default method, you can override it.

- Any call to print() on an SLList will use this method instead of default.
- Use (optional) @Override to catch typos like **`public void pirnt()`**

```
public interface SLList<Item> implements  {
    @Override
    public void print() {
        for (Node p = sentinel.next; p != null; p = p.next) {
            System.out.print(p.item + " ");
        }

        System.out.println();
    }
}
```

itastructur.es

Question

Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y.

Which print method do you think will run when the code below executes?

- List.print()
- SLList.print()

```
public static void main(String[] args) {
    List61B<String> someList = new SLList<String>();
    someList.addLast("elk");
    someList.addLast("are");
    someList.addLast("watching");
    someList.print();
}
```

datastructur.es

Question

Recall that if X is a superclass of Y, then an X variable can hold a reference to a Y.

Which print method do you think will run when the code below executes?

- List.print()
- **SLList.print() : And this is the sensible choice. But how does it work?**
  - Before we can answer that, we need new terms: static and dynamic

```
public static void main(String[] args) {
   List61B<String> someList = new SLList<String>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   someList.print();
}
```

datastructur.es

Static and Dynamic Type, Dynamic Method Selection

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

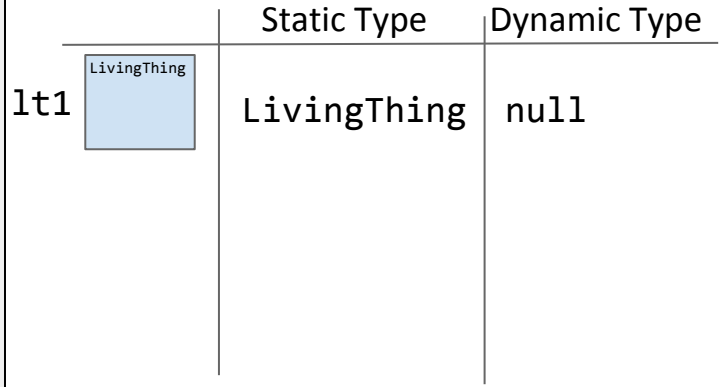
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



datastructur.es

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

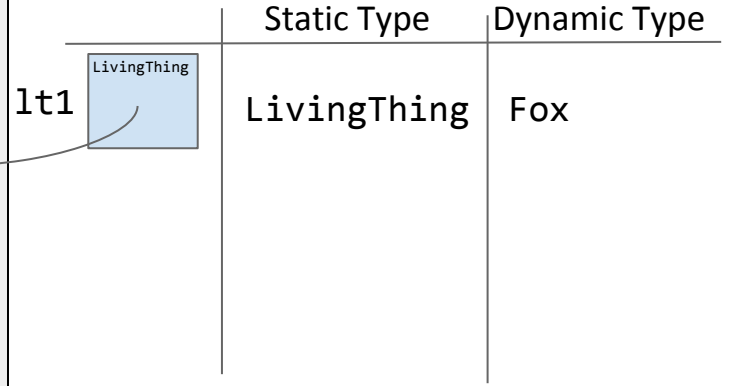
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = new Fox();
   lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



datastructur.es

Static Type vs. Dynamic Type

Every variable in Java has a “compile-time type”, a.k.a. “static type”.

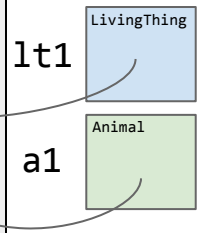
- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```

Technically requires a “cast”. See next lecture.



	Static Type	Dynamic Type
lt1	LivingThing	Fox
a1	Animal	Fox

datastructur.es

Static Type vs. Dynamic Type

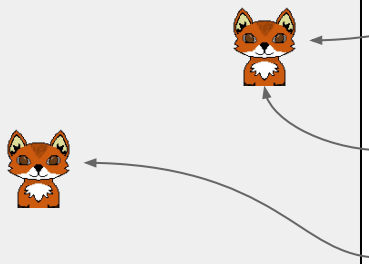
Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```



	Static Type	Dynamic Type
lt1	LivingThing	Fox
a1	Animal	Fox
h1	Fox	Fox

datastructur.es

Static Type vs. Dynamic Type

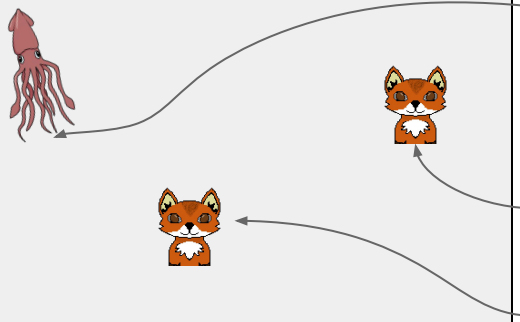
Every variable in Java has a “compile-time type”, a.k.a. “static type”.

- This is the type specified at **declaration**. Never changes!

Variables also have a “run-time type”, a.k.a. “dynamic type”.

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
    LivingThing lt1;
    lt1 = new Fox();
    Animal a1 = lt1;
    Fox h1 = new Fox();
    lt1 = new Squid();
}
```



	Static Type	Dynamic Type
lt1	LivingThing	Squid
a1	Animal	Fox
h1	Fox	Fox

datastructur.es

Dynamic Method Selection For Overridden Methods

Suppose we call a method of an object using a variable with:

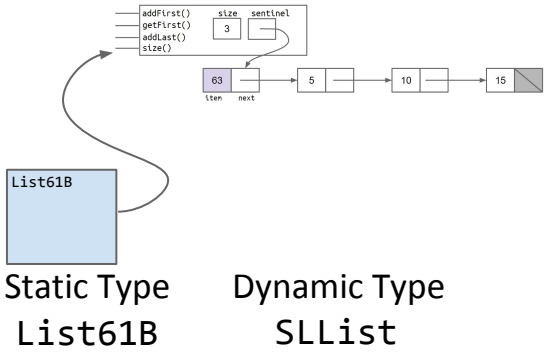
- compile-time type X
- run-time type Y

Then if Y **overrides** the method, Y's method is used instead.

- This is known as "dynamic method selection".

This term is a bit obscure.

```
public static void main(String[] args) {
   List61B<String> s1= new SLList<String>();
   someList.addLast("elk");
   someList.addLast("are");
   someList.addLast("watching");
   someList.print();
}
```



datastructur.es

More Dynamic Method Selection, Overloading vs. Overriding

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);
a.sniff(d);
d.flatter(d);
a.flatter(d);
```

datastructur.es

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);   // "hello animal"
a.sniff(d);
d.flatter(d);
a.flatter(d);
```

datastructur.es

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);   // "hello animal"
a.sniff(d);   // "dog sniff animal"
d.flatter(d);
a.flatter(d);
```

datastructur.es

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);   // "hello animal"
a.sniff(d);   // "dog sniff animal"
d.flatter(d); // "u r cool dog"
a.flatter(d);
```

datastructur.es

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.greet(d);   // "hello animal"
a.sniff(d);   // "dog sniff animal"
d.flatter(d); // "u r cool dog"
a.flatter(d); // “u r cool animal”
```

`flatter` is overloaded, not overridden!

The Method Selection Algorithm

Consider the function call `foo.bar(x1)`, where `foo` has static type `TPrime`, and `x1` has static type `T1`.

At compile time, the compiler verifies that `TPrime` has a method that can handle `T1`. It then records the signature of this method.

- Note: If there are multiple methods that can handle `T1`, the compiler records the "most specific" one. For example, if `T1=Dog`, and `TPrime` has `bar(Dog)` and `bar(Animal)`, it will record `bar(Dog)`.

At runtime, if `foo`'s dynamic type overrides the recorded signature, use the overridden method. Otherwise, use `TPrime`'s version of the method.

We did not see this in our lecture puzzle, but see study guide for more!

datastructur.es
CC BY NC SA

Dynamic Method Selection Puzzle

Suppose we have classes defined below. Try to predict the results.

```
public interface Animal {
  default void greet(Animal a) {
    print("hello animal"); }
  default void sniff(Animal a) {
    print("sniff animal"); }
  default void flatter(Animal a)
    print("u r cool animal"); }
}
```

```
public class Dog implements Animal {
  void sniff(Animal a) {
    print("dog sniff animal"); }
  void flatter(Dog a) {
    print("u r cool dog"); }
```

```
Animal a = new Dog();
Dog d = new Dog();
a.flatter(d);
```

Compiler asks “Is there a method in `Animal` that can handle `Dog`? Yes! `flatter(Animal a)`”. It then records the signature `flatter(Animal a)`.

datastructur.es

Interface vs. Implementation Inheritance

Interface vs. Implementation Inheritance

Interface Inheritance (a.k.a. what):

- Allows you to generalize code in a powerful, simple way.

Implementation Inheritance (a.k.a. how):

- Allows code-reuse: Subclasses can rely on superclasses or interfaces.
  - Example: print() implemented in List61B.java.
  - Gives another dimension of control to subclass designers: Can decide whether or not to override default implementations.

Important: In both cases, we specify “is-a” relationships, not “has-a”.

- Good: Dog implements Animal, SLList implements List61B.
- Bad: Cat implements Claw, Set implements SLList.

datastructur.es
CC BY NC SA

The Dangers of Implementation Inheritance

Particular Dangers of Implementation Inheritance

- Makes it harder to keep track of where something was actually implemented (though a good IDE makes this better).
- Rules for resolving conflicts can be arcane. Won't cover in 61B.
  - Example: What if two interfaces both give conflicting default methods?
- Encourages overly complex code (especially with novices).
  - Common mistake: Has-a vs. Is-a!
- Breaks encapsulation!
  - What is encapsulation? See next week.

datastructur.es
CC BY NC SA

Terminology Summary

New terms from this lecture:

- Overloading
- Hypernym
- Hyponym
- Overriding
- Interface Inheritance
- Implementation Inheritance
- Static Type, a.k.a. Compile-time Type
- Dynamic Type, a.k.a. Run-time Type
- Dynamic Method Selection

datastructur.es