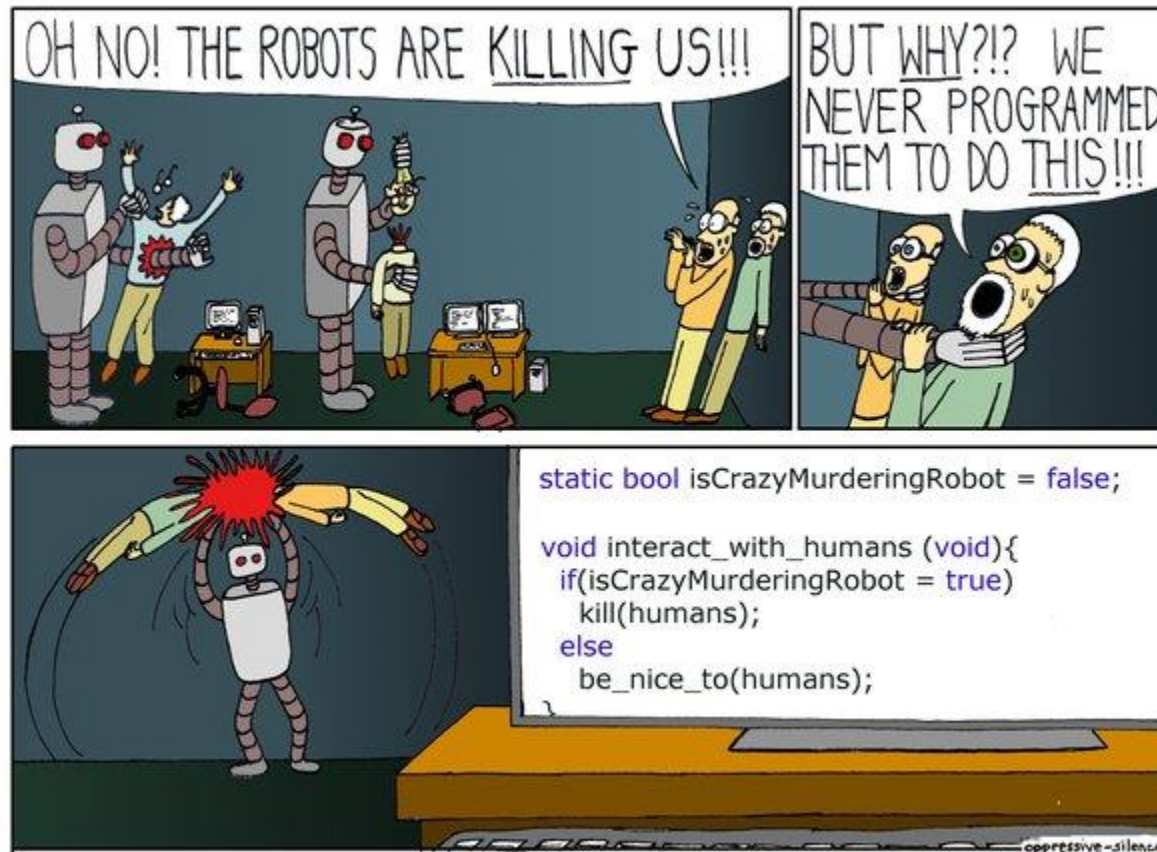


CS 142

Lecture 22: interfaces; grammars



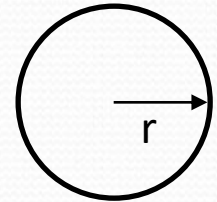
Thanks to Marty Stepp and Stuart Reges for parts of these slides

Related classes

Consider classes for shapes with common features:

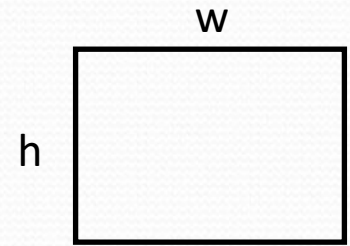
- Circle (defined by radius r):

$$\text{area} = \pi r^2, \quad \text{perimeter} = 2 \pi r$$



- Rectangle (defined by width w and height h):

$$\text{area} = w h, \quad \text{perimeter} = 2w + 2h$$

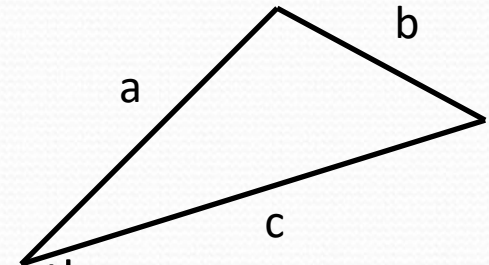


- Triangle (defined by side lengths a , b , and c)

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = \frac{1}{2}(a+b+c)$,

$$\text{perimeter} = a + b + c$$



- Every shape has these, but each computes them differently.

Storing Multiple Types

- It would be nice if we could store shapes of different types in a list. What do we need in order to do this?

- Are there any downsides?

Solution: Interfaces

- **interface:** A list of methods that a class can promise to implement.
 - Inheritance gives you an is-a relationship *and* code sharing.
 - A `Lawyer` can be treated as an `Employee` and inherits its code.
 - Interfaces give you an is-a relationship *without* code sharing.
 - A `Rectangle` object can be treated as a `Shape` but inherits no code.
 - Analogous to non-programming idea of roles or certifications:
 - "I'm certified as a CPA accountant.
This assures you I know how to do taxes, audits, and consulting."
 - "I'm 'certified' as a Shape, because I implement the Shape interface.
This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {  
    public type name(type name, ..., type name);  
    public type name(type name, ..., type name);  
    ...  
    public type name(type name, ..., type name);  
}
```

Example:

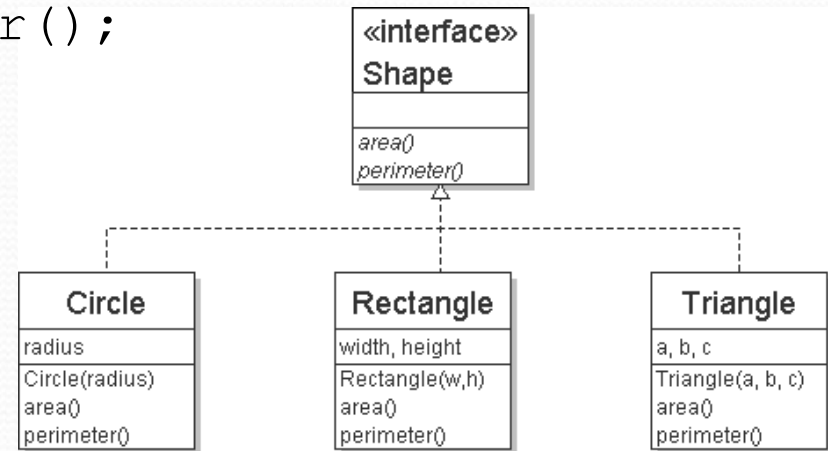
```
public interface Vehicle {  
    public int getSpeed();  
    public void setDirection(int direction);  
}
```

Shape interface

// Describes features common to all shapes.

```
public interface Shape {  
    public double area();  
    public double perimeter();  
}
```

- Saved as Shape.java



- **abstract method:** A header without an implementation.
 - The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.

Implementing an interface

```
public class name implements interface {  
    ...  
}
```

- A class can declare that it "implements" an interface.
 - The class must contain each method in that interface.

```
public class Bicycle implements Vehicle {  
    ...  
}
```

(Otherwise it will fail to compile.)

Banana.java:1: Banana is not abstract and does not
override abstract method area() in Shape

```
public class Banana implements Shape {  
    ^
```

Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
 - They allow **polymorphism**.
(the same code can work with different types of objects)

```
public static void printInfo(Shape s) {  
    System.out.println("The shape: " + s);  
    System.out.println("area : " + s.area());  
    System.out.println("perim: " + s.perimeter());  
    System.out.println();  
}
```

...

```
Circle circ = new Circle(12.0);  
Triangle tri = new Triangle(5, 12, 13);  
printInfo(circ);  
printInfo(tri);
```

Lists

- We implemented an `ArrayIntList` class so we could create a list that changed size as data was added

index	0	1	2	3
value	42	-3	17	9

- Is there any other way we could have created a list structure?

Abstract data types (ADTs)

- **abstract data type (ADT):** A specification of a collection of data and the operations that can be performed on it.
 - Describes *what* a collection does, not *how* it does it
- We don't know exactly how other types of lists are implemented, and we don't need to.
 - We just need to understand the idea of the list and what operations it can perform.

An IntList interface

// Represents a list of integers.

```
public interface IntList {  
    public void add(int value);  
    public void add(int index, int value);  
    public int get(int index);  
    public int indexOf(int value);  
    public boolean isEmpty();  
    public void remove(int index);  
    public void set(int index, int value);  
    public int size();  
}
```

```
public class ArrayIntList implements IntList { ...
```

```
public class LinkedIntList implements IntList { ...
```

Client code w/ interface

```
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);

        IntList list2 = new LinkedIntList();
        process(list2);
    }

    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1);
        System.out.println(list);
    }
}
```

Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<>();
```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:

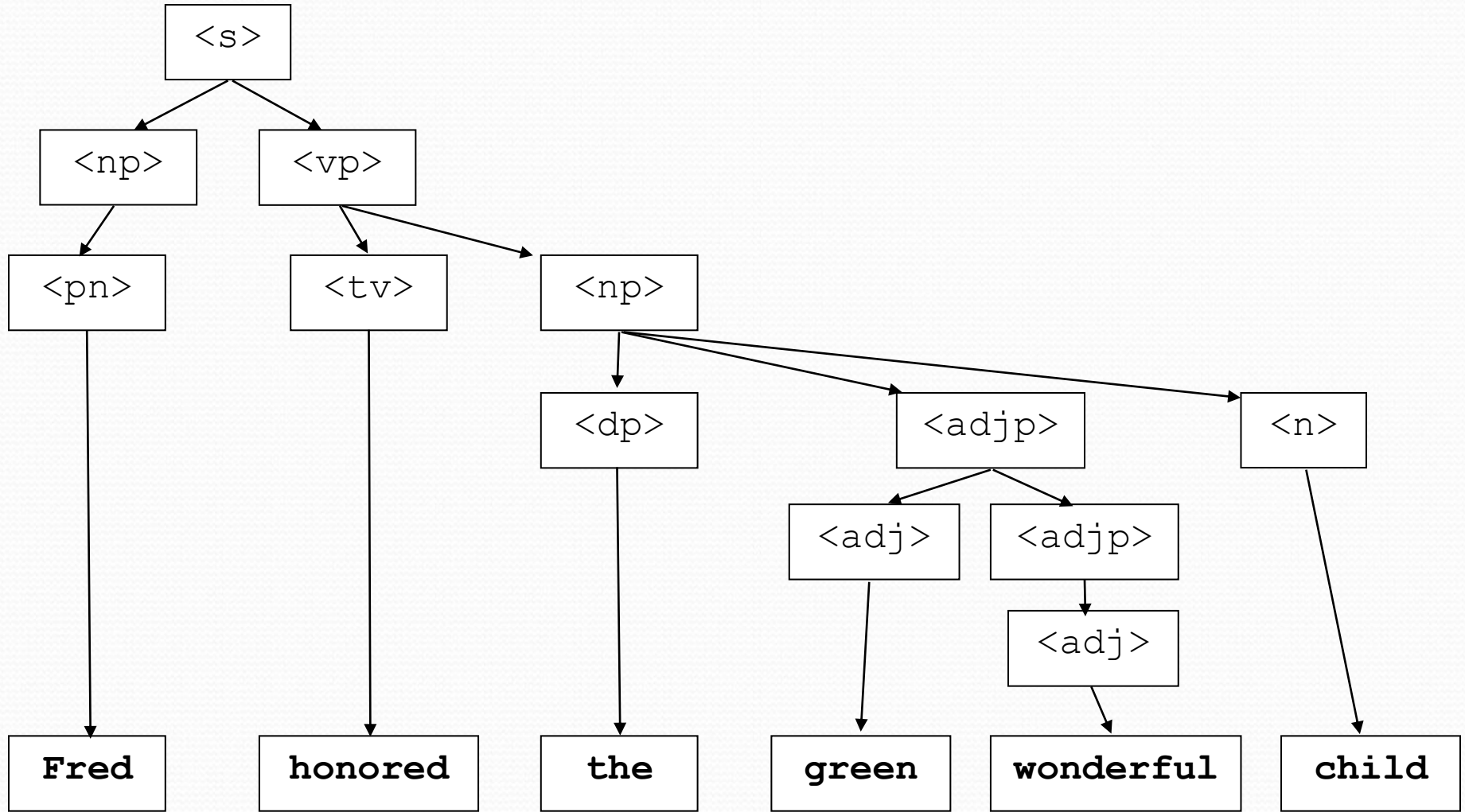
```
public void stutter(List<String> list) {  
    ...  
}
```

Languages and Grammars

Languages and grammars

- (formal) **language**: A set of words or symbols.
- **grammar**: A description of a language that describes which sequences of symbols are allowed in that language.
 - describes language *syntax* (rules) but not *semantics* (meaning)
 - can be used to generate strings from a language, or to determine whether a given string belongs to a given language

Sentence generation



Backus-Naur (BNF)

- **Backus-Naur Form (BNF):** A syntax for describing language grammars in terms of transformation *rules*, of the form:

<symbol> ::= <expression> | <expression> ... | <expression>

- **terminal:** A fundamental symbol of the language.
- **non-terminal:** A high-level symbol describing language syntax, which can be transformed into other non-terminal or terminal symbol(s) based on the rules of the grammar.
- developed by two Turing-award-winning computer scientists in 1960 to describe their new ALGOL programming language

An example BNF grammar

`<s> ::= <n> <v>`

`<n> ::= Marty | Allison | Stuart | Jessica`

`<v> ::= cried | slept | belched`

- Some sentences that could be generated from this grammar:

Marty slept

Jessica belched

Stuart cried

BNF grammar version 2

`<s> ::= <np> <v>`

`<np> ::= <pn> | <dp> <n>`

`<pn> ::= Marty | Allison | Stuart | Jessica`

`<dp> ::= a | the`

`<n> ::= ball | hamster | carrot | computer`

`<v> ::= cried | slept | belched`

- Some sentences that could be generated from this grammar:

the carrot cried

Jessica belched

a computer slept

BNF grammar version 3

`<s> ::= <np> <v>`

`<np> ::= <pn> | <dp> <adj> <n>`

`<pn> ::= Marty | Victoria | Stuart | Jessica`

`<dp> ::= a | the`

`<adj> ::= silly | invisible | loud | romantic`

`<n> ::= ball | hamster | carrot | computer`

`<v> ::= cried | slept | belched`

- Some sentences that could be generated from this grammar:

the invisible carrot cried

Jessica belched

a computer slept

a romantic ball belched

Grammars and recursion

`<s> ::= <np> <v>`

`<np> ::= <pn> | <dp> <adjp> <n>`

`<pn> ::= Marty | Victoria | Stuart | Jessica`

`<dp> ::= a | the`

`<adjp> ::= <adj> <adjp> | <adj>`

`<adj> ::= silly | invisible | loud | romantic`

`<n> ::= ball | hamster | carrot | computer`

`<v> ::= cried | slept | belched`

- Grammar rules can be defined *recursively*, so that the expansion of a symbol can contain that same symbol.
 - There must also be expressions that expand the symbol into something non-recursive, so that the recursion eventually ends.

Grammar, final version

`<s> ::= <np> <vp>`

`<np> ::= <dp> <adjp> <n> | <pn>`

`<dp> ::= the | a`

`<adjp> ::= <adj> | <adj> <adjp>`

`<adj> ::= big | fat | green | wonderful | faulty | subliminal`

`<n> ::= dog | cat | man | university | father | mother | child`

`<pn> ::= John | Jane | Sally | Spot | Fred | Elmo`

`<vp> ::= <tv> <np> | <iv>`

`<tv> ::= hit | honored | kissed | helped`

`<iv> ::= died | collapsed | laughed | wept`

- Could this grammar generate the following sentences?

Fred honored the green wonderful child

big Jane wept the fat man fat

- Generate a random sentence using this grammar.