1

MORE JAVA!

Lecture 3
CS2110 Spring 2013

Recall from last time

2

- We were thinking about a toy store
  - Everything on the shelves is a "toy"
  - A toy has a price, a suggested age range, a name (it might be different in different countries…), a weight
  - Yet not all kinds of toys are the same
    - A "My Little Pony" and a "GI Joe Action Figure" have atttributes that differ
    - A pony has a cute color. A GI Joe has a weapon…

Toy.java

3

- A class to capture the main features of toys. Let's focus on toys that all "have a button"

```
/** A class representing toys that have a button */
public class Toy {
    private String name;              // Culture-specific name
    private int ageLow, ageHigh;   //  For children aged…
    public Toy(string name, int ageLow, int ageHigh) { …}
    public void pushTheButton() { …}
}
```

Let's define our three methods

4

- First method is the constructor

```
public Toy(String name, int ageLow, int ageHigh) {
    this.name = name;
    this.ageLow = ageLow;
    this.ageHigh = ageHigh;
}
```

- Why **this**? Permits reuse of the same names for the parameters to the constructor as for the fields.
  - Not needed if same names hadn't been reused
  - **this** is a name for the "object in which it appears"

Use of "this"

5

```
/** A class representing toys that have a button
 */
public class Toy {
    private String name;
    private int ageLow, ageHigh;
    public Toy(String name, int ageLow, int ageHigh) {
        this.name = name;
        this.ageLow = ageLow;
        this.ageHigh = ageHigh;
    }
}
```

Rules for accessing fields

6

- For object t1,
  - t1 = **new** Toy("ActionWarrior", 10, 15) creates new object and invokes constructor for class Toy with these arguments
  - t1.PushTheButton() invokes method push-button for the instance of Toy that t1 currently points to
  - Many classes define the same method a few times with different parameters. This is called **polymorphism**
    - t1.PushTheBottom(howHard);
- Without an object reference, Java looks for the "closest" plausible match, from "inside to outside".
  - This is why "ageLow" resolves to the parameter ageLow, whereas this.ageLow resolves to the instance variable.

Why didn't Toy have a return type?

7

- The Toy constructor has no return type

  Tells Java that Toy is a method for initializing new instances of the class. Called in a new-expression

- PushTheButton has return type **void**

  Tells Java it is a a method that does something, but doesn't return anything.

Instance methods

8

- PushTheButton is an "instance" method
  - It appears in each object of class PushTheButton
  - It "sees" the instance variables
    - Each Toy object has its own values for name, ageLow, …
    - The instance method can access those values
    - Thus t1.PushTheButton() sees the values of these variables for object t1. t2.PushTheButton() sees values for t2.

Inherited methods

9

- In Java, every class is a *subclass* of some other class
  - The classes we just showed you are subclasses of class Object, which is the *superclass* for all objects
  - Object has some methods
    - toString(), Equals(), ….
- A subclass can redefine the methods it inherits from its parent class; next lecture shows this
  - Right now, t1.toString() would return a string containing the type Toy and the name t1.
  - But we could redefine toString(), e.g. to produce a string reflecting the name of the Toy, and the age range…

Static versus instance

10

- Recall that main was static. In fact we can mark any method or any field as static.
- static means: there is just one version shared by all instances of this type of object
  - When calling such a method or accessing such a variable, you don't give an object instance.
  - A static method can't "see" instance methods or fields of an object unless you specify the object instance

Use of static: Assign an id to each toy

11

```
public class Toy {
    private String name;
    private int ageLow, ageHigh, myId;
    private static int nextId= 0; // first unassigned id
    public Toy(string name, int ageLow, int ageHigh) {
        this.name = name;
        this.ageLow = ageLow;
        this.ageHigh = ageHigh;
        myId = nextId++;
    }
}
```

Things to notice

12

- Could have written **this.**myId but didn't need **this** because myId was unambiguous
  - myId references an instance field in current Toy object
- nextId is shared by all Toy objects

myId = nextId++;	*means*	myId = nextId; nextId = nextId+1;
myId = ++nextId;	*means*	nextId = nextId + 1; myId = nextId;

The idea of an « abstraction »

13

- Early in the exploration of computing, Von Neuman pointed out that a sufficiently powerful digital computer can "simulate" any other digital computer
  - The computer as an *infinitely specializable* machine
- A computer can "compute" in novel ways
  - For example, your laptop hardware has no idea what a "file" is, or that cmm.mpeg contains "Call me maybe"
  - Yet we think of the computer as knowing that this is a music file. It understands operations like "seek" or increase volume"

Computational Thinking

14

- With languages like Java we can express very sophisticated "ideas" through objects
- The key is to think about the object as if it really was a music video, or a graphical representation of the routes from Ithaca to Key West, or a Tweet
  - Object captures necessary data to represent something
  - And has thing-specific operations, like "play"
- Languages to help us program in this abstracted way have hugely amplified our power as computer scientists

A concrete example of abstraction

15

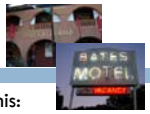
- Think about driving instructions in Google Maps. How would you program with "routes" in Java?
  - A route is basically a graph: a sequence of nodes (locations) linked by edges (roads)
  - So… We might imagine a "class" representing graphs
  - It would use classes representing nodes, edges
  - Graph operations like *shortest path* used to solve problems like recommending the best route home

But wait!

16

- Nodes could be:
  - Intersections, cities, hotels, motels, restaurants, gas stations, amazingly awesome tourist attractions, places where road construction is happening, radar detectors…
- Edges could be:
  - Highways, toll roads, carpool-only express lanes, small roads, dirt roads, one-way roads, bridges, draw-bridges, ferries, car-trains, tunnels, seasonal roads…
- We might want a separate type for each, but how would we write code to find a route in a map?

Google maps: Our goal



17

- We want to be able to write code like this:

```
foreach Node nd in the route
    if(nd is a Hotel, and nd has 3 or more stars)
        println("Why not check out " + nd + " ?");
```

- E.g. look at a route node by node, and for each node check to see what type it is, and then print a list of the 3-star or better hotels.
- … but not the 3-star balls of twine. And road intersections probably don't have stars, even in Yelp!

Core of the dilemma

18

- On the one hand we want our route to include many kinds of map-route nodes.
  - They differ because they represent different things
  - Hotels have swimming pools and bars, while road intersections have round-abouts and traffic lights
- … and sometimes we just say "the route goes from here, to here, to here" node by node
- … but other times, we need to be type-specific

Extending a class

19

- Java also has a feature for taking some class and adding features to create a new class that is a "specialized version" of it
  - Called "creating a subclass" by "extending" the parent
  - Very similar to the idea of implementing an interface with one major difference
    - When you extend a parent class, your class "inherits" all the fields and methods already defined by the parent class
    - You can add new ones but the old ones are available
    - You can also redefine ("override") the old ones. We'll see why this can be useful later

From Lodging to Hotel

20

- It would be natural for all the various "lodging" options to share certain methods
- This way we know that every lodging on a route can be described in the same way, checked for how many stars it has, etc
- Then we can introduce specialized subclasses for the subtypes of the parent type

A problem...

21

- One issue that now arises is that sometimes we want a single object to behave like more than one kind of parent object
- In Java this is not permitted
  - Any given class "extends" just one other class
  - If not specified, this will automatically be "Object"

Interfaces

22

- An **interface** is a class that defines fields and methods but in which the methods aren't filled in
  - We specify the method names and parameter types but omit the body – they are "abstract"
- Java allows classes to implement multiple interfaces
  - As we'll see, any class has one definition, but that definition can implement multiple interfaces
  - To implement an interface the class has to tell Java it is going to do so, and has to implement all its methods

... so in Java

23

- We have notations to
  - Create a new subtype from a parent type, overriding its definitions if desired
  - Define interfaces, and then define types that implement those interfaces
- We'll look closely at how these look, and work

Notation

24

```
class Hotel extends Lodging
                implements MapNode, Comparable
{
    … fields and methods specific to Hotel
    … fields and methods that implement MapNode
    … fields and methods to implement Comparable
}
```

What might the Lodging parent class have?

25

- Address, phone number, "how many stars"
  - … basically, fields and methods that all lodgings share.
  - A lodging is a place to stay, but only some of them have bedrooms, and only some have camping sites
- So Lodging is a category covering things shared by all forms of lodgings.
  - When designing a parent class, you try and factor out common functionality shared by the subclasses

A Hotel is a kind of lodging

26

- It has single rooms, suites, maybe a swimming pool
- Many are parts of chains like Hilton, Ramada…
- So a Hotel can be defined as a subclass of Lodging
  - It would add additional fields and methods that other kinds of lodgings don't necessarily support
  - Eg, "bed size" isn't relevant for a campground
  - Wooded/Meadow aren't relevant for a hotel

Extending a class: Details

27

- If class B **extends** class A, B is a **subclass** of A and A is a superclass of B.
  - Subclass B can add new fields and methods.
  - You can treat an object of class B as an object of class A — it is "both at the same time".
    - E.g. a Hotel object is also a Lodging object, and all of them are Objects. No casting is required to access parent methods

A method inherited from a parent

28

- Suppose that you do create class Lodging. It out will automatically support **toString()**
  - Why? Because Lodging extends Object
  - … and the Object superclass defines toString()
  - … so every kind of object inherits a toString() method!
- Of course, you might not like that version of toString()
  - By default, toString() prints name@MemoryLocation
  - … but you can **override** toString() and replace it with a version that generates some other kind of string, like…

"Hotel CaliforniaYou can check out any time you like, but you can never leave."

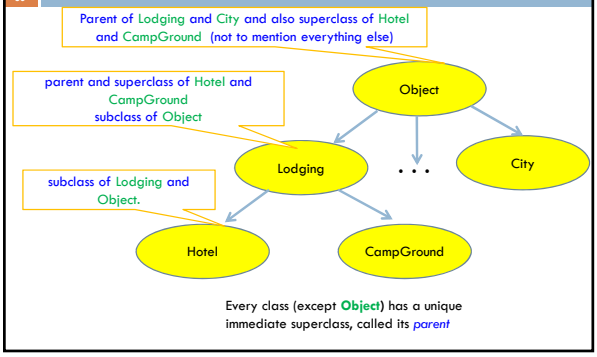
Extends versus implements

29

- An interface is a kind of empty class
  - It defines fields and methods, but with method bodies
  - Java knows the type signatures for the methods
- If a class **implements** an interface, it needs to "fill in the blanks" by providing code for all interface methods
  - Because those methods had no bodies, this is different than extending a parent by adding new fields and methods, or overriding a method defined in a parent class.
  - We treat the object as if it had the Interface type

Class Hierarchy

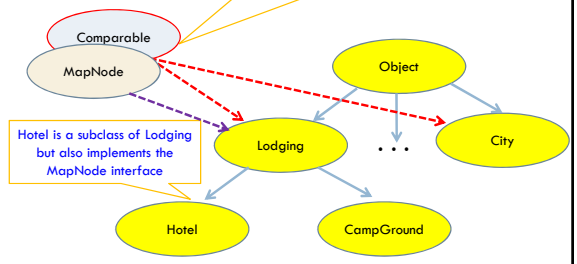
30



Interfaces

31

Interfaces are "fully abstract" classes listing "type signatures" for fields and methods. A a class implementing the interface is required to define all of those methods but each can define them in its own specialized way.



A class has only one parent in the type hierarchy, but can implement many interfaces. You make the decision of how you want the class hierarchy to look, and what interfaces to support as part of a process aimed at having clean, elegant code

Back to Google maps…

32

- We could say that a node is any class supporting an interface with operations like sellsGas(), hasFood(), …
- An edge implements an interface too: it can represent roads, bridges, ferries but always has properties like isTollRoad(), speedLimit(), expectedTime(timeOfDay)…
- Then we can write map code that can find a route that includes many types of nodes and edges along it!

Google maps: Our goal

33

```
foreach Node nd in the route
    if(nd is a Hotel, and nd has 3 or more stars)
        println("Why not check out " + nd + " ?");
```

Google maps

- For example if Route is a a list of nodes this code could be used to print all the possible 3-star hotels:

```
List<MapNode> route = map.getRoute("San Jose", "Ithaca");
for(MapNode nd: route)
    if(nd instanceof Hotel && ((Hotel)nd).YelpStars >= 3)
        println("Why not check out " + nd + " ?");
```

- … so this code looks at each node on the route, and for those that are hotels, prints 3-star ones
- List<MapNode> is a "generic"… we'll discuss soon

Google maps

35

- For
could be used to print

This "cast" says that we're sure that nd refers to a Hotel and want to treat it that way. The cast would fail if nd was some other kind of object implementing the MapNode interface

Check that the node is a Hotel, not something like a gas station or an intersection

```
for(MapNode nd: Route)
    if(nd instanceof Hotel && ((Hotel)nd).YelpStars >= 3)
        println("Why not check out " + nd + " ?");
```

- … so this code looks at each node on the route, and for those that are hotels, prints 3-star ones

Using the class hierarchy

36

- With the class hierarchy we can write very general styles of Java code
  - Programs that have some code that handles general collections of objects that could of very different types but that all support the same interface
  - Other code might be highly specialized for just one object type
- This flexibility helps us avoid needing to rewrite the same code more than once

instanceof

37

- Operator **instanceof** can peer into the type of an object

```
if(a instanceof Type)
    ((Type)a).someMethod();
```

- **null instanceof** Type is false
- Type is either a class name or an interface name
- Java assumes that a has its declared type.
  - If this creates any ambiguity, we **cast** a to the appropriate type. We did that here because not every Route node is a Hotel object.
  - The cast will fail, throwing an exception, if a isn't an object of the desired type. For example, you can't cast a Hotel to a CampGround because neither is a subclass of the other in our class hierarchy.
  - No cast is required if Java can unambiguously determine the method.

Overloading of Methods

38

- A class can have several methods of the same name
  - … each having a different *parameter signature*
  - The *parameter signature* of a method is its name plus the types of its parameters. (The return type isn't included)
- Example: We might want two constructors for the Hotel class, one for cheap hotels and one for fancy hotels that have a swimming pool
  - Hotel elCheapo = new Hotel("Bates Motel", 3);
  - Hotel deluxe = new Hotel("Miami Hilton", 300, "pool");
- Usually the versions with fewer parameters just call the ones with more parameters, plugging in default values

Overloading is often used to support default parameter values

39

- … this is incredibly common
  - There are many settings where "fancy" users of an object specify values for things that "basic" users might omit
  - You want to specify defaults: "no swimming pool", or "no bar"
  - The best approach is to with *all possible parame* that offer convenient subs
- For example:

```
public void RingTheBell() { RingTheBell(Limply); }
public void RingTheBell(Vigor howHard) { …. }
```

One of the possible Vigor values

This overload has no parameters

Probably an "enumerated" type

Illustrating this idea: **Overloads** of "toString"

40

- The Object class defines the default "toString()"
- There are predefined overloads of toString() to print the common object types: Integer, String, Double, etc.
- Java will automatically call toString() if conversion to a string seems to be what you've requested, as in:

```
println("Why not check out " + nd + " ?");
```

  - This code automatically calls nd.toString(), concatenates the result to end up with one string, then calls println()