CSSE2002/7023

Programming in the large

Week 3.1: More inheritance and things going wrong

More inheritance

`class Child extends Parent` $\Rightarrow$
`Child` gets the following from `Parent`

1. (public/protected) methods and variables. (What is says it can do.)
2. Method bodies/implementation. (How it does it)

Sometimes all we care about is what public methods are present in a class. (Not trying to inherit code) Only #1.

interfaces

An `interface` is a type like a class but it contains no method bodies[1].

Eg: `java.lang.Comparable`
declares a single method, `compareTo` which takes an object and returns an `int` $\in \{< 0, 0, > 0\}$.

So any code which takes objects and will need to order them, could specify them has being of type `Comparable`.

Eg: `int doStuff(Comparable c[])`

[1] "What about default from Java9?" - Shhhh

interfaces

To declare that your class complies with an interface use `implements`.

```
public class Duck extends Fowl
        implements Comparable, Clonable
```

This says that `Duck` inherits code and members from the `Fowl` class *and* has all the methods which `Comparable` and `Clonable` say should be there.

interfaces

- A class implementing an interface is responsible for supplying method bodies for everything declared in the interface.
- Could be used to advertise that your class has additional useful capabilities.
- Could be used to indicate that your class belongs to multiple groups. (eg: Someone is both a staff member and a student?)
- `interface`s fill a role taken by abstract[2] classes in other languages. It is necessary because Java only allows extending from a single class[3].

[2] which Java also has

[3] "single inheritance" - Python allows multiple inheritance

super

`Square1.java`

Use of `super`, shadowing and `this`

No Ducks

Python will let you write code to access a member without knowing whether it actually exists. Instead it checks at runtime.
Java won't let you try to access something unless it is sure (at compile time) that it exists.

```
Object ob="Hello";
int l1=ob.length(); // compile error
                    // Object has no .length()
```

```
Object ob="Hello";
String s1=(String)ob;
int l1=s1.length();
```

OR

```
Object ob="Hello";
int l1=((String)ob).length();
```

Casting

`(newtype)oldvalue` is called a "typecast" or just "cast". It tells the Java that you want a *newtype* value of *oldvalue*.

```
(int)1.7
long x=100; (float)x
```

Some casts will be made automatically:

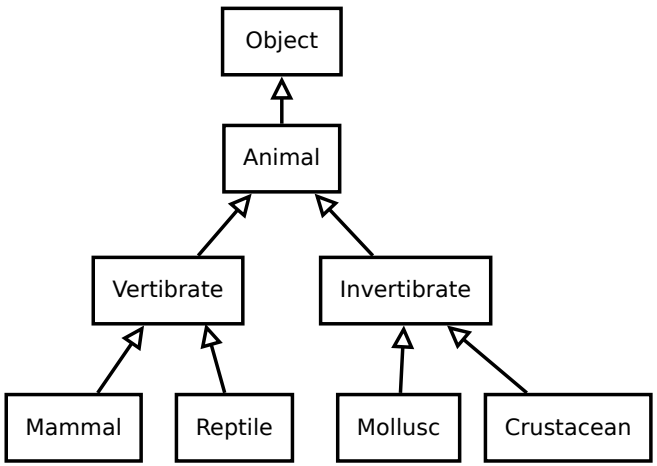
- "smaller" types $\rightarrow$ "larger" types. (eg `int` to `long`, `float` to `double`).
- integer types $\rightarrow$ floating point types (eg `int` to `float`).

Casting a variable does not change the value of the variable.

Casting references

`(String)ob`
will give a reference which can be used to access `String` members in `ob`. That is, it changes the compiler's view of what it is looking at. It does not change `ob` itself. If the compiler does not know of an inheritance relationship between the known type of `ob` and the cast type, it will not compile.
At runtime, java will check to see if the cast is valid. If it isn't, it will throw a `ClassCastException`.

Casting references



From	To	Implicit?	Error?
Vertibrate	Animal	yes	—
Vertibrate	Mammal	no	runtime?
Vertibrate	Object	yes	—
Vertibrate	Mollusc	no	compile

"Upcasts" can be implict, "Downcasts" can't.

You cannot cast between primatives and objects.

Eg: `(String)5;` is no bueno.

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code
- `boolean` return

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code
- `boolean` return
- error code return

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code
- `boolean` return
- error code return
  - `int` indicating what went wrong. [Use static constants for this]

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code
- `boolean` return
- error code return
  - `int` indicating what went wrong. [Use static constants for this]
  - Make sure you include one for success

When it all goes wrong

Breaking code into functions means we need to work out what to do if something fails.

- Things can fail? Surely I'd notice.
- Silent failure — detect, and avoid doing something silly
- Print to console
  - Fine for quick debugging
  - Hard to process from code
- `boolean` return
- error code return
  - `int` indicating what went wrong. [Use static constants for this]
  - Make sure you include one for success
  - Variants: String, enums

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.
  - eg: indexOf() == -1 or position; Object or `null`

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.
  - eg: indexOf() == -1 or position; Object or `null`
  - Be careful that the sentinel is never a valid value. (See Y2K)

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.
  - eg: indexOf() == -1 or position; Object or `null`
  - Be careful that the sentinel is never a valid value. (See Y2K)
  - May need a separate method to get details of the failure.

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.
  - eg: indexOf() == -1 or position; Object or `null`
  - Be careful that the sentinel is never a valid value. (See Y2K)
  - May need a separate method to get details of the failure.
- Return a status object containing results or failure information.

Need to get an answer out as well?

- Sentinel — A special value which is not meant to be taken literally.
  - eg: indexOf() == -1 or position; Object or `null`
  - Be careful that the sentinel is never a valid value. (See Y2K)
  - May need a separate method to get details of the failure.
- Return a status object containing results or failure information.
- *Extra information out via OUT or reference params (Not applicable to Java)*

Everything so far has used the return value to get info out.

Exceptions

`exc1.java`

- Don't just squash exceptions
- Once an exception has been thrown, it will unwind the stack until caught. Return does not happen.
- `finally` happens whether or not an exception was caught
- Trigger an exception with `throw`.
- A `try` can have multiple `catch` blocks