Java/OO recap

Background

A typical Java program can consist of many *classes*

- Object-oriented programming has been found to be an excellent approach to software production

Object modelling is used to help us design such systems

- A good bridge between the real world problem and the software solution
- Classes are effective representations of kinds of problem domain entities
- And classes can become components of the software solution

Although we are concentrating on analysis and design, we will start by looking at object-oriented *programming in Java* to keep a clear connection between design and implementation

- Also a good way to understand the meaning of the UML object modelling concepts and notations

Object-Orientation in Java

The text of an object-oriented program consists of a set of *class definitions*

A *class* defines:

- The *attributes* (variables, data) of an *object* (usually *private*),
- The public *operations* (methods) "offered" by the object – for call from other objects,
- The *behaviour* of the object when one of its operations is called,
- Any supporting *private methods*

An *object* is an *instance of a class*

- A class is like a *template*, and an *object* is like a copy of the template
- Created using Java's **new** keyword, eg:

```
StaffRecord sr = new StaffRecord("Simon Jones");
```

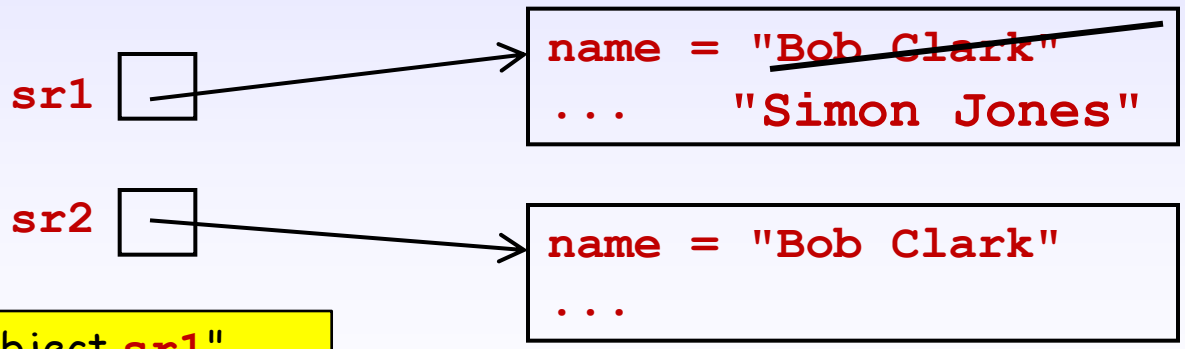
At run-time, a Java program consists of a group of *communicating objects* which are *created* from a set of *class definitions*

Each object has a separate physical identity

- Concretely: Each object occupies a *fresh block of RAM* - and variables such as **sr** hold *references* to objects
- Its attributes might change value, but *it remains the same object*
- Two objects may have the same set of attribute values, but are *two distinct objects if created with separate* **new**

Example:

```
StaffRecord sr1 = new StaffRecord("Bob Clark");
StaffRecord sr2 = new StaffRecord("Bob Clark");
sr1.setName("Simon Jones");
sr1 = sr2;  ???
```



Note: "the object **sr1**" means "the object referred to by variable **sr1**"

Note: the *references* in **sr1**, **sr2** are typically the *RAM addresses* of the objects they *refer* to

About *constructors*:

In order to be able to have:

```
new StaffRecord("Bob Clark")
```

The class `StaffRecord` must have a *constructor*:

```
public StaffRecord(String name) {
    this.name = name;
}
```

Note: reminder about `this`

Reminder: When a `new` instance of a class is constructed:

1. Fresh memory (RAM) is allocated
2. The constructor is called
3. A reference to the new object (in RAM) is returned

A class may have:

- No explicit constructor (there is an implicit "no-args" constructor)
- One constructor (could be "no-args" or have parameters)
- *Several* constructors – *overloading*
  Examples: `JButton()`, `JButton(Icon i)`, `JButton(String text)`

About *overloading:*

Any class may have *more than one method* with the *same name*, provided that:

- The formal parameters lists are *distinguishable*
- This allows the compiler/JVM to determine *which actual method* is being called

Examples:

```
JButton b1 = new JButton();
JButton b2 = new JButton("Press me");
```

If `StaffRecord` has, say:

```
public void setDetails(String room) {...}
public void setDetails(String room, String telNo) {...}
```

then

```
sr1.setDetails("4B63");
sr1.setDetails("4B63","7434");
```

Reminder: All classes implicitly *extend* the `Object` superclass

Two important methods are inherited from `Object` :

`equals` `toString`

equals: `public boolean equals(Object o)`

- Called like this: `if ( o1.equals(o2) )`
- Like `o1 == o2` this checks whether the two references are identical (that is *references to the same object*)
- It is often more useful to check whether the two objects have "equivalent contents" - `equals` must be *overridden*
- Example for `StaffRecord`:

```
public boolean equals(Object o) {
  return name.equals((StaffRecord)o.name);
}
```

Note: the cast

Note: the `String` `equals` method

- Then can call it like this:

```
if ( sr1.equals(sr2) )
```

- Note: `sr1 == sr2` is meaningful but not always appropriate

toString: `public String toString()`

- Called like this: `System.out.println( o.toString() );`
  or, equivalently: `System.out.println( o );`
- `toString` returns: the name of the class of the object `@` a hexadecimal hash code for the object (eg `Student@33c0d9d`)
- It is normal to *override* `toString` to display something meaningful for our own classes
- Example for `StaffRecord`:

```
public String toString() {
  return name + " in room " + room;
}
```

- Then can call it like this:

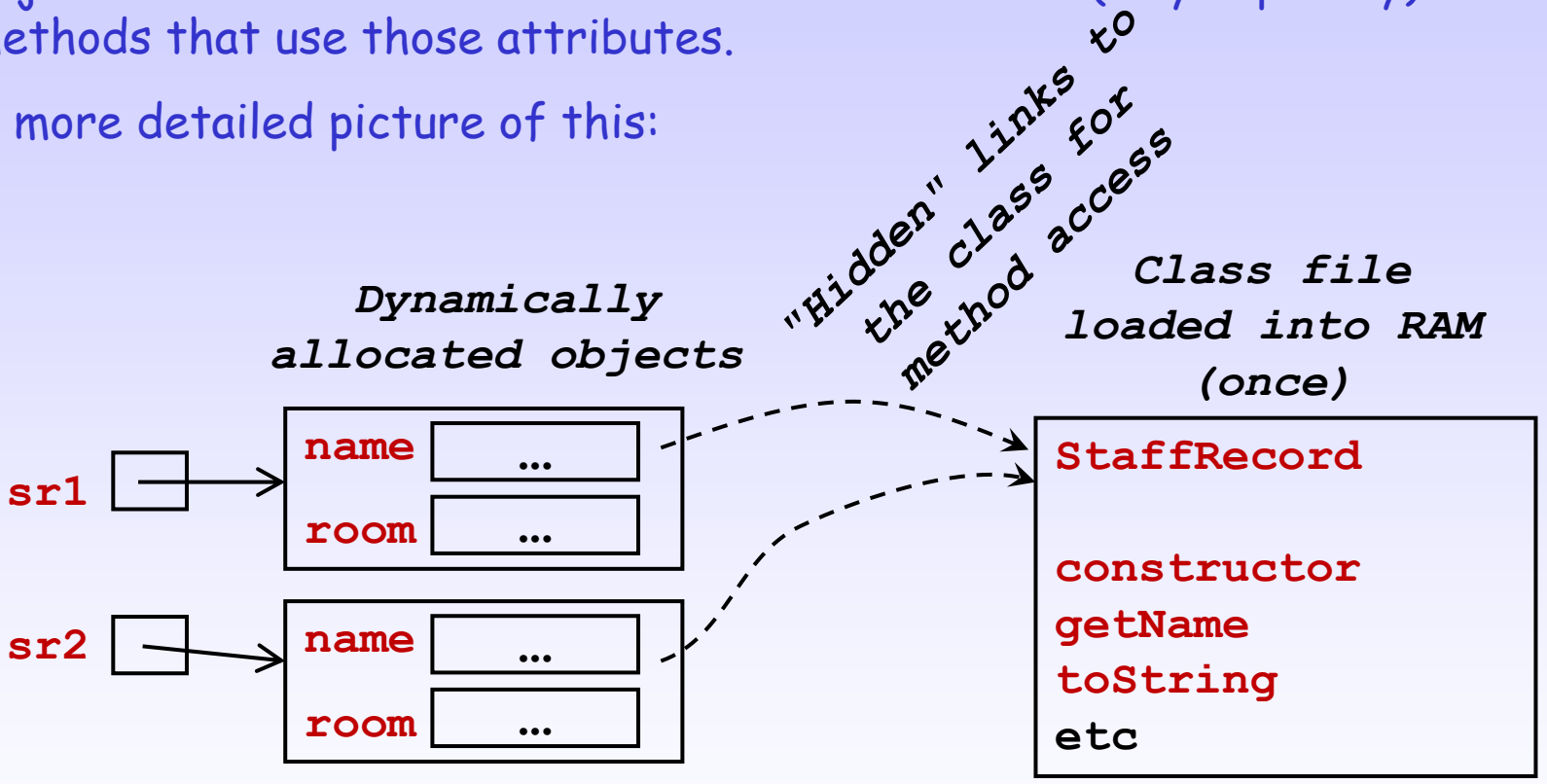
```
textfield.setText( sr1.toString() );
System.out.println("Info: " + sr1);
```

Note: + gives implicit `toString`

Static versus Dynamic:

The picture so far is purely *dynamic*: Classes are instantiated to give objects which contain *their own attribute data* and (only implicitly) methods that use those attributes.

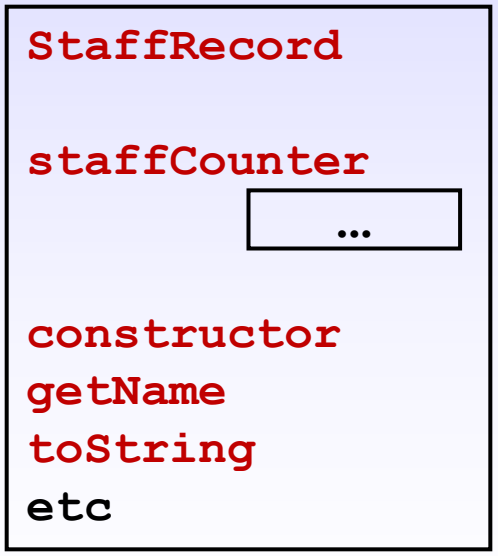
A more detailed picture of this:



Now the *static* aspects:

We can also store variable/attribute data **in the class RAM itself**.

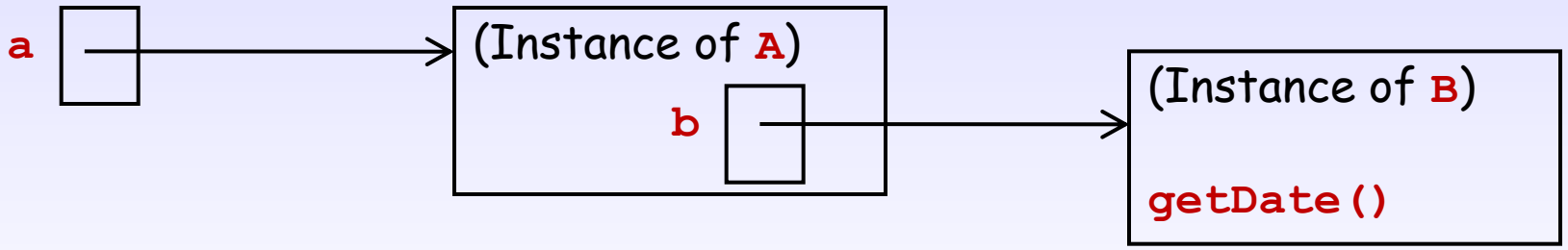
- These are called "class variables" (or attributes) and are **not** duplicated when a new object is allocated.
- *All instance objects* **share** the class variables.
- Declared with the keyword `static`. Example in `StaffRecord`:
  `private/public static int staffCounter = 0;`
- Then in the constructor we could have:
  `staffCounter++;  id = staffCounter;`
  – a shared count of all instances, and unique ids!
- Can be referred to via objects:
  `sr1.staffCounter` `sr2.staffCounter`
- Or via the class name:
  `StaffRecord.staffCounter`
  even without any instances having been allocated!
- Can also have `static` *methods* – can only refer to `static` variables, not dynamic (think about it!)
- **Use with care!**



Consider an object **a** (an instance of class **A**) and an object **b** (an instance of class **B**)

A **client** object **a** communicates with a **server** or **supplier** object **b** by calling an *operation* "offered" by object **b**

- We often think of **a** as sending a message to **b**
- **a** may be sending information to **b**, requesting information from **b**, requesting that **b** carry out some action, or a combination
- Client object **a** must contain a reference to the supplier object **b**
- In Java, we usually talk about *methods* rather than operations



The client **a** has a task it needs to perform e.g. calculate days expired:

```
Date d = b.getDate();
```

where **a** has previously created an instance of **b**:

```
B b = new B(...);
```

The supplier **b** provides a service that helps **a** with its task e.g. get the current date:

```
public Date getDate() {…}
```

Inheritance

In inheritance, a ***subclass*** "extends" the definition of its ***superclass***

- It implicitly includes the attributes and operations from the superclass, and may add attributes and operations and may redefine the implementations of operations

For example, inheritance is shown in the following, where classes **`Book`** and **`Journal`** "inherit from" **`Publication`**: (full text on next slide)

```
public class Publication{ ... }

public class Book extends Publication { ... }

public class Journal extends Publication { ... }
```

We can have an inheritance ***hierarchy***

- If, say, **`Publication`** is itself a subclass of an even higher-level class
- Or, say, **`Book`** has subclasses

```
public abstract class Publication {
  private int catNum;                          (later protected)
  private String title;                        (later protected)
  public String getTitle() { ... }
  public int getCatNum() { ... }
  public abstract void borrow(Member m);
  public void return() { ... }
}

public class Book extends Publication {
  private String author;
  public String getAuthor() { ... }
  public void borrow(Member m) { ... }
}

public class Journal extends Publication {
  private int volNum;
  public int getVol() { ... }
  public void borrow(Member m) { ... }
}
```

Inheritance & Attributes

Although we want to hide the attributes of a superclass from ordinary clients, we usually want to make them *visible in the subclass*

- This is *not automatic* with **`private`** attributes

Hence, we there are *three levels of visibility*: public, private and protected.

- **`public`** : visible to clients
- **`private`** : visible only within the class
- **`protected`** : visible only within the class *and its subclasses*

So that the method bodies in **`Book`** and **`Journal`** can refer to **`title`** and **`catNum`**:

```
protected int catNum;
protected String title;
```

Inheritance & Operations

Note that the operation **borrow** is given in *all three classes* **Publication**, **Book** and **Journal**

- That shows that it is defined in **Publication** ...
- And **redefined** or **overridden** in the **Book** and **Journal** subclasses

This is useful if we wish to indicate:

- That all **Publications**, whether they are a book or a journal, have a public **borrow** operation
- But that the *actual details* for borrowing a **Book** and borrowing a **Journal** are *different* – so the two subclasses must have separately defined methods

We also need to decide whether we need an actual *implementation* (method body) of **borrow** in **Publication**

- There does *not need to be*, and in this example there is not – indicated by the keyword **abstract**
- (See discussion of "abstract methods" later on)

Instantiation of subclasses

To be completely clear:

When a *subclass* is instantiated using **`new`**:

- Enough storage is allocated for *all the attributes* of the subclass and its one (or more) superclasses
- Effectively, all the subclass's operations are included, ***plus*** all the ***non-overridden operations*** from the superclasses **except** the constructors
- A subclass constructor may call its superclass's constructor using

```
super(…);
```

- (This is Java; details may vary slightly for other OO languages)

Therefore:

- An instance of a subclass *has all the properties expected of an instance of a superclass*
- For example: a **`Book`** has **`title`**, **`getTitle`**, **`catNum`**, **`return`**, **`borrow`**, etc

A consequence of the previous slide, and a feature of inheritance in object-oriented systems is that:

- Anywhere a reference to a superclass object is expected, we can use a reference to a subclass object instead
- … because we can guarantee that it has all the capabilities

Hence, a reference to a **`Publication`** object could refer to a **`Book`** object

For example, in Java, we can have the declaration:

```
Publication pub = new Book("UML");
```

- The type of **`pub`** is reference to a **`Publication`**, but **`pub`** is *currently referring to* a **`Book`** object

But this is *definitely **not** allowed:*

```
Book book = new Publication(...);
```

- Because the **`Publication`** object does not have all the properties expected of a **`Book`** object

Further:

- When we call an operation through a reference, the *actual method used* depends on the *actual object referred* to

(Again, this is Java, and details may vary in other OO languages)

For example, in Java, after the declaration:

```
Publication pub = new Book("UML");
```

- If we call the `borrow` operation: `pub.borrow(...);`
  it is the method defined in the `Book` class that is used and not the one defined in `Publication`.

- If `pub` were currently referring to a `Journal` object and we call `pub.borrow();` it would be the method defined in the `Journal` class that would be used

These features are known as **polymorphism** and **dynamic binding**

Try reading:
http://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Inheritance & Abstract classes

Sometimes, it does not make sense to have an instance of the superclass:

- The superclass is then being used only to define attributes and operations that are common to all its subclasses
- Such a superclass is called an **abstract class**

We can indicate that **`Publication`** is to be an **abstract class**, i.e. one which has no instances and is therefore only there to be inherited from:

```
public abstract class Publication { ... }
```

Although, we cannot have (direct) instances (objects) of an abstract class such as **`Publication`**, we can have variables that can hold references to **`Publication`** objects:

- At run-time, the references will actually refer to objects of a *subclass* of **`Publication`**.

Inheritance & Abstract methods

We can also have **abstract methods** or operations which have a heading, but no body

- For example the method `borrow` in class `Publication` is an `abstract` method
- A `borrow` method **must then be defined** in both the `Book` and `Journal` subclasses
- *or* in any further subclass which is itself to be non-abstract and instantiable

A class *must* be `abstract` if it has one or more `abstract` methods

But we can also *require* that a class is `abstract` even if none of its methods are `abstract`

Try reading:
http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Interfaces

Some programming languages support *multiple inheritance* where a subclass may have *more than one* superclass

- This is available in the implementation language C++
- But ***not*** in Java – it avoids various complications

Java only has *single inheritance*

- But it also has **`interface`**`s`

Interfaces provide the *advantages* of multiple inheritance *without the disadvantages*.

Try reading:
http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

Interfaces & Abstract Classes

An **abstract class**

- May have attributes
- And some of its operations may have implementations

An **interface** is like a class

- But has **no attributes** (except `final` constants)
- And *none* of its operations have implementations

An interface is therefore like a *very abstract class*

- It simply lists the public operations that an "extending" class ***must provide implementations for***
  (but we use `implements` rather than `extends`)
- In effect it summarises a set of capabilities, a "contract"

A class that offers actual methods for the public operations "promised" by an interface is said to implement that interface

Why interfaces are useful

Interfaces are useful as they allow our designs/programs to be more general/flexible than they otherwise would be

Natural choices for interface names are ***adjectives***

For example:

- We can use an interface name, say `Moveable`, as the *type of a formal parameter*:

```
private void myMethod(Moveable m, ...) { ... }
```

- This indicates that `myMethod` is happy to receive *any object at all* as an actual parameter *provided* that it offers the operations specified in `Moveable`
- The actual parameter can be an instance of *any class* that "`implements Moveable`" (and the Java compiler checks for us!)

Interfaces - Example

What is going on might become clearer if we look at some Java:

```
public interface Moveable {
   public void left(int d);
   public void up(int d);
}

public class Rectangle implements Moveable {
   ... left ... up ... (full definitions)
   ... size ... grow ... (Rectangle specific items) ...
}

public class Balloon implements Moveable {
   ... left ... up ... (full definitions)
   ... expand ... (Balloon specific items) ...
}
```

And we could have a *very general class* **Mover** that has a reference to a **Moveable** as an attribute, and a main program that uses it:

```
class Mover {
    Moveable m;
    public Mover(Moveable m) {
      this.m = m;
    }
    private void moveIt() {
      m.left(17);
      m.up(25);
    }
```

Main program:

```
mR = new Mover(
        new Rectangle(...));
mB = new Mover(
        new Balloon(...));
mR.moveIt();
mB.moveIt();
```

m can refer to *any object that implements the* **Moveable** *interface*

- However, the only methods that can be called are those offered by **Moveable**
- So, **even** when **m** is pointing at a **Rectangle** object, it *cannot call* the **Rectangle** operations **size** and **grow**

End of lecture