Topic 9: Combining Objects

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 1st December 2016

programming@kcl.ac.uk
martinchapman.co.uk/teaching

Q: So far in the course, how have we combined objects?

Preface: Has-A Relationships

COMBINING OBJECTS: ASSOCIATION RELATIONSHIPS (1)

So far, we've already seen one technique we can use to **combine** objects in order to **model** entities:

- We declare a field in a class (Class A) to be of another **class type** (Class B), and then store objects of Class B in Class A.

- Objects of Class A use objects of Class B to **define parts of their own functionality**.

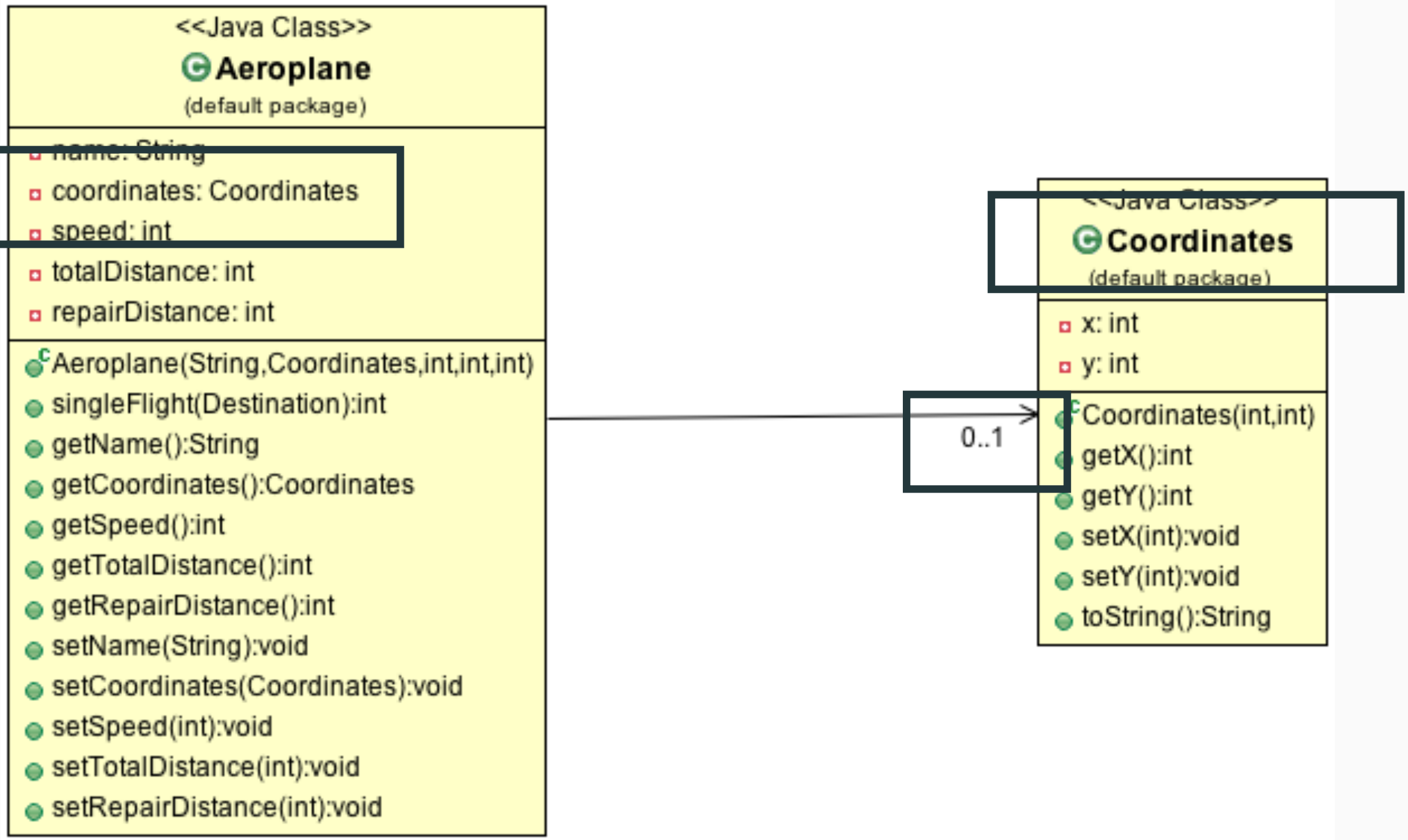
- Consider an `Aeroplane` object that uses a `Coordinates` object to store its position.

COMBINING OBJECTS: ASSOCIATION RELATIONSHIPS (2)

```
public class Aeroplane {

   private String name;
   private Coordinates coordinates;
   private int speed;
   private int totalDistance;
   private int repairDistance;

   public Aeroplane(String name, Coordinates coordinates,
            int speed, int totalDistance, int repairDistance) {

      this.name = name;
      this.coordinates = coordinates;
      this.speed = speed;
      this.totalDistance = totalDistance;
      this.repairDistance = repairDistance;

   }
```

We say that an Aeroplane is **associated with** a pair of Coordinates.

REPRESENTING ASSOCIATION RELATIONSHIPS IN CLASS DIAGRAMS (1)



You were not expected to include this relationship in your documentation to date, as we hadn't yet covered it during the lectures.

REPRESENTING ASSOCIATION RELATIONSHIPS IN CLASS DIAGRAMS (2)

To represent an association relationship in a class diagram, we draw a **solid line** from Class A (the class using objects of Class B as a part of its own representation) to Class B.

- Until now we have only used **dashed** lines to represent a subset of **dependency**: when Class A makes an object of Class B.

On this line, we place two numbers, to represent **multiplicity**.

- The first number represents the **lower bound** (the **least** number of objects of Class B there can be stored in Class A).
- The second number (following the three dots) represents the **upper bound** (the **maximum** number number of Class B objects there can be in Class A).
- Naturally therefore, this will always be **0...1** unless a field represents a **list** of objects (here a ***** represents **no fixed upper bound**).

OBJECT OWNERSHIP (1)

Association relationships often imply that objects are combined by **passing one object into another**, perhaps from a **driving** class.

- For example, in CW2, we created our Dishes in CalorieTracker, and passed them to a Meal object.

```
Dish toast = new Dish();
toast.setCalories(140);
Meal omeletteBreakfast = new Meal();
omeletteBreakfast.setStarter(toast);
```

- This gave us the flexibility to use the **same** Dish in different Meals.

REMEMBER: OBJECTS IN MEMORY (3)

That's because, **at our current level of abstraction**, the rule is slightly different for objects.

When an object is passed to a method, any interactions with that object **will alter the original object.**

- **Reassigning the variable holding the class copy**, however, will **not** alter the **object**.

Next semester, when we look at objects in memory (i.e. **a lower level of abstraction**), the reasoning behind this should become clearer.

😴

OBJECT OWNERSHIP (2)

Because manipulating an object in one place in your program affects **all other uses** of that object, implementing an associative relationship may result in unexpected behaviour:

```
Coordinates destination1Coordinates = new Coordinates(110,
                                                      135);
Destination destination1 = new Destination("Buenos Aires",
                                           destination1Coordinates);

Aeroplane plane = new Aeroplane("Spirit",
                                destination1Coordinates, 12, 0, 2550);
```

Here, when the plane flies, so will Buenos Aires!

OBJECT OWNERSHIP (3): COMPOSITION

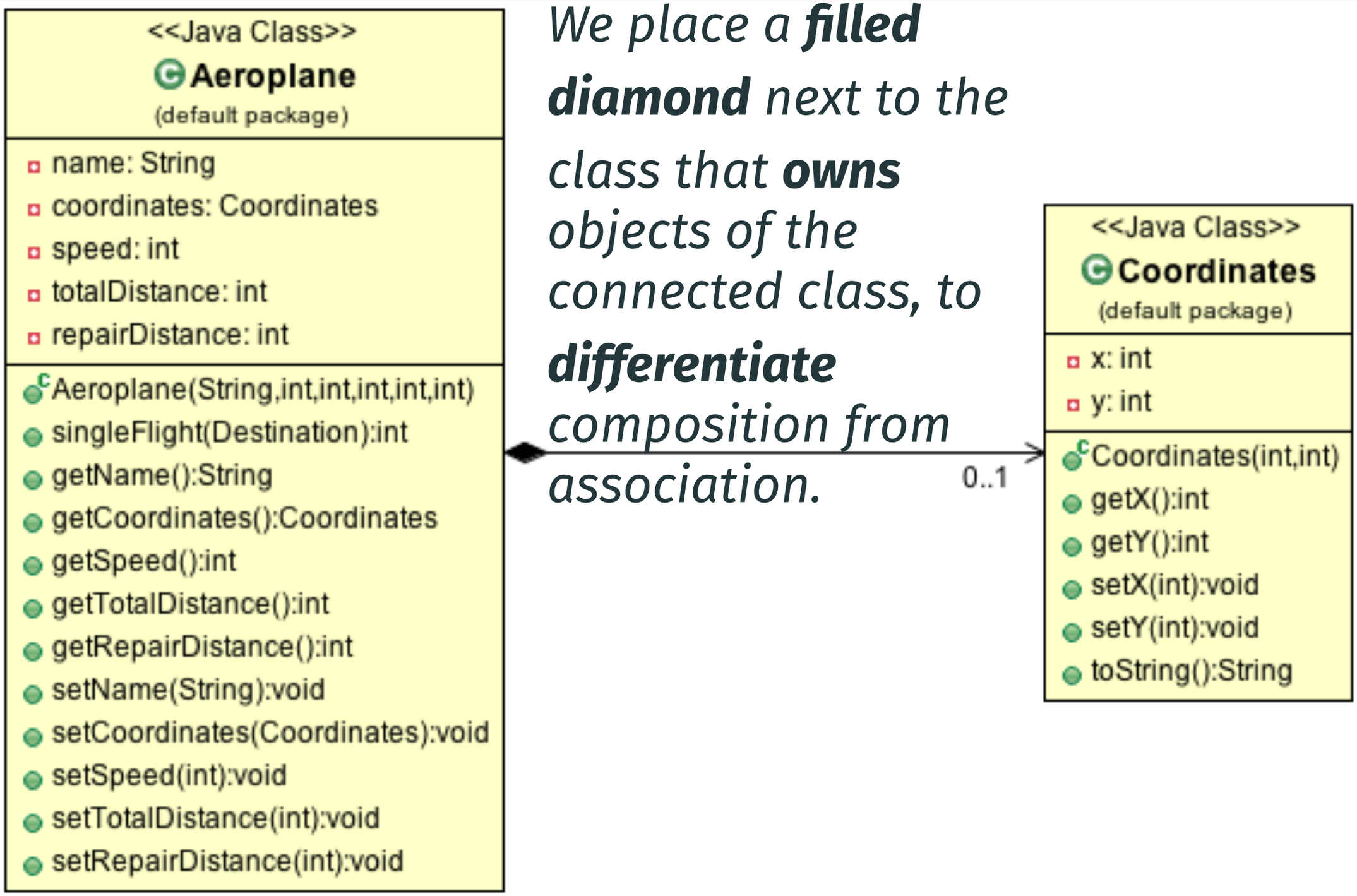
Instead, sometimes we want instances of Class A to **own** all the instances that is **uses** of Class B. This avoids the danger of **global updates**.

- All the instances of Class B that are used by Class A only **exist** within instances of Class A.

- This is referred to as a **composition** relationship.

```
public class Aeroplane {
   ...
   private Coordinates coordinates;
   ...
   public Aeroplane(..., int x, int y, ...) {
      ...
      this.coordinates = new Coordinates(x, y);
      ...
   }
```

REPRESENTING COMPOSITION RELATIONSHIPS IN CLASS DIAGRAMS



*We place a **filled diamond** next to the class that **owns** objects of the connected class, to **differentiate** composition from association.*

You were not expected to include this relationship in your documentation to date, as we hadn't yet covered it during the lectures.

HAS-A RELATIONSHIPS

We can generalise the idea of both **association** and **composition** to the single idea of an object **having** an instance of another.

- We call these **HAS-A** relationships.

OBJECTIVES: COMBINING OBJECTS

In this topic, to complement HAS-A relationships, we will look at another way to combine objects, or for one class to define part of its functionality via another:

- **Inheriting from a class** such that (an object of) one class **IS-A** (object of an)other class.
- **Specialising** the inherited class, for **a new purpose**.

We will also:

- **Compare** IS-A and HAS-A relationships.
- Discuss the different **forms** of IS-A relationships.

Students often find this the hardest topic in PPA.

Modelling A Medieval Village

A MEDIEVAL VILLAGE

Let’s imagine a medieval village that has a number of **villagers**, who have the capability to **fight** in order to defend their village from attacks by **trolls**, and how we might model this.

A VILLAGER

A VILLAGER: IN CODE

```
public class Villager {

   private String name;

   public Villager(String name) {

      this.name = name;

   }

   public String getAttack() {

      return name + " throws a punch";

   }

   public String getName() {

      return name;

   }

}
```

A villager has a name, and an attack.

However, weedy villagers can't attack too well, so there is no **damage value** associated with this attack.

Instead, we represent this by simply returning a string.

A SWORDSMAN

A SWORDSMAN: IN CODE (FIRST PASS)

```
public class Swordsman {

   private String name;

   public Swordsman(String name) {

      this.name = name;

   }

   public String getAttack() {

      return name + " swings a sword";

   }

   public String getName() {

      return name;

   }

}
```

A swordsman is just another **type** of villager, except an **armed** one, but they too are **ineffective**.

Because a class's **identity** is defined by its **interface** (i.e. its public methods), we might choose to represent this by creating a class with the **same** interface.

But really, we should try and avoid this **repetition**, as it makes our code difficult to **maintain**. What are our **options**?

FLEXIBLE OBJECTS (1)

Recall in CW4 we saw that an exam can play **two different roles** (as a mark scheme or as an exam attempt).

```
Exam markScheme = new Exam(nqMarkScheme, bqMarkScheme, mcqMarkScheme, 7);
Exam examAttempt = new Exam(nqAttempt, bqAttempt, mcqAttempt, 0);
```

Can we apply the same idea here to avoid writing a second class?

```
public class Village {

    public static void main(String[] args) {

        Villager villager = new Villager("Villager");
        Villager swordsman = new Villager("Swordsman");

    }

}
```

FLEXIBLE OBJECTS (2)

Whereas an exam could easily represent **either** a mark scheme **or** a student script because both the interface and the **functionality** behind the interface are the **same**, the swordsman's functionality differs slightly, so using the same class is difficult:

```
public String getAttack() {

   return name + " throws a punch";

}
```

Villager.java

```
public String getAttack() {

   return name + " swings a sword";

}
```

Swordsman.java

What we really want is for Villager and Swordsman to share the **same interface** but **vary** the **functionality behind that interface**.

AWKWARD HAS-A RELATIONSHIPS (1)

```
public class Swordsman {

    private Villager villager;

    public Swordsman(String name) {

        this.villager = new Villager(name);
                Composition makes sense here to
    }           retain unique individuals.

    public String getAttack() {

        return villager.getName() + " swings a sword";

    }

    public String getName() {

        return villager.getName();

    }

}
```

We could, if we wanted to, use a HAS-A relationship in an attempt to **alter the functionality behind the interface**, while **minimising repetition**.

We **delegate** the task of storing and retrieving the name to the existing functionality in the `Villager` object, while adding to the functionality.

AWKWARD HAS-A RELATIONSHIPS (2)

While the use of a HAS-A relationship here allows us to alter the functionality behind the interface, we still have **duplicate** methods in each class that perform the same functionality (e.g. `getName`).

Part of this issue comes from the fact that with a HAS-A relationship we are only able to express when things **own** other things not when they **are** other things, and thus share a common interface.

- A `Swordsman` **IS-A** `Villager` it does not **have** (HAS-A) villager (like `Aeroplane`'s have `Coordinates`), which is also awkward **conceptually**.

In other words, we want our two classes to **literally** share the same interface.

How do we do this?

CLASS INHERITANCE (1)

```
public class Villager {

    private String name;

    public Village
        this.name =

    }

    public String

        return name

    }

    public String

        return name

    }

}
```

```
public class Swordsman extends Villager {

    public Swordsman(String name) {

        super(name);

    }

    public String getAttack() {

        return getName() + " swings a sword";

    }

}
```

CLASS INHERITANCE (2)

One class **inherits** (**copies**) **all of the code** from another, thus assuming the same **interface**.

Because a class's interface determines its identity, the inheriting class is now **also** considered to be of the **same type** as the class it inherits from.

- We therefore achieve our desired **IS-A** relationship.

CLASS INHERITANCE (3)

```
public class Village {

   public static void main(String[] args) {

      Swordsman swordsman = new Swordsman("Swordsman");
      System.out.println(swordsman.getName());

   }

}
```

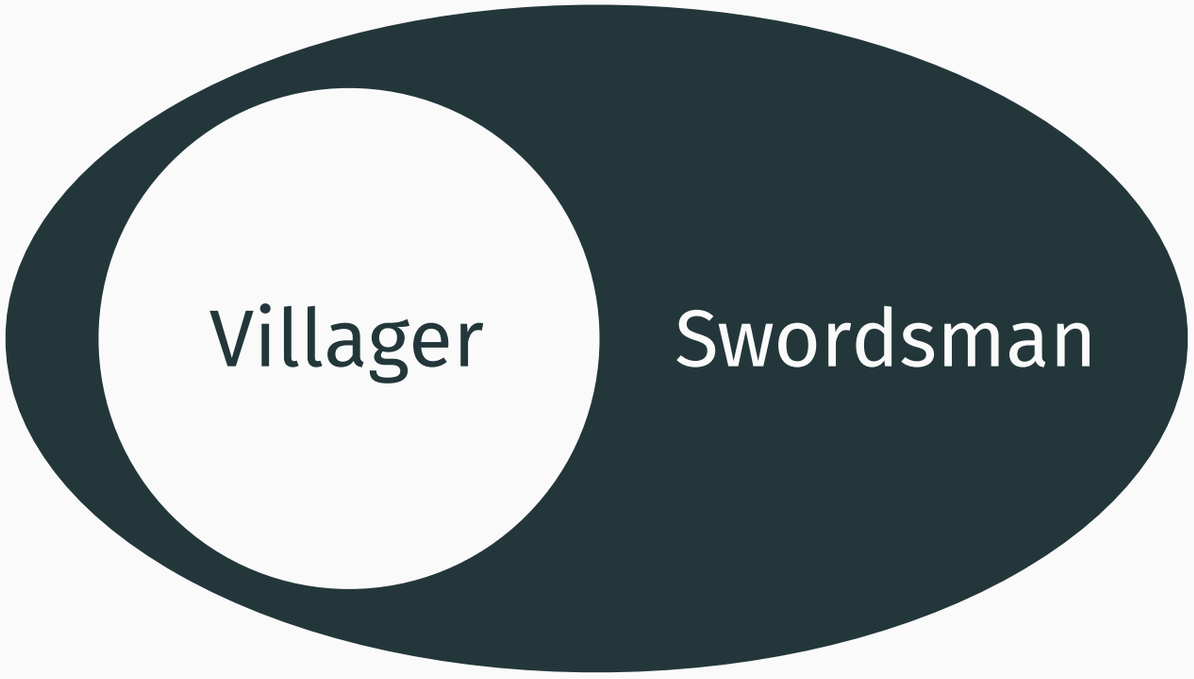
Although the `getName` *method doesn't exist in* `Swordsman`*, we can still call it, because a* `Swordsman` *is now also a* `Villager`*, and thus has the same interface (the same public methods), and the associated fields in which to store data.*

CLASS INHERITANCE (4)

Moreover, while **retaining** this interface, the inheriting class can **specialise** the behaviour behind the interface.

- We can **redefine** the behaviour in each method.

The inheriting class can also **add to** the interface.



*The inheriting class thus shares **at least** the same interface as the class it inherits.*

Let’s examine the notion of class inheritance in more detail.

THE EXTENDS KEYWORD (1)

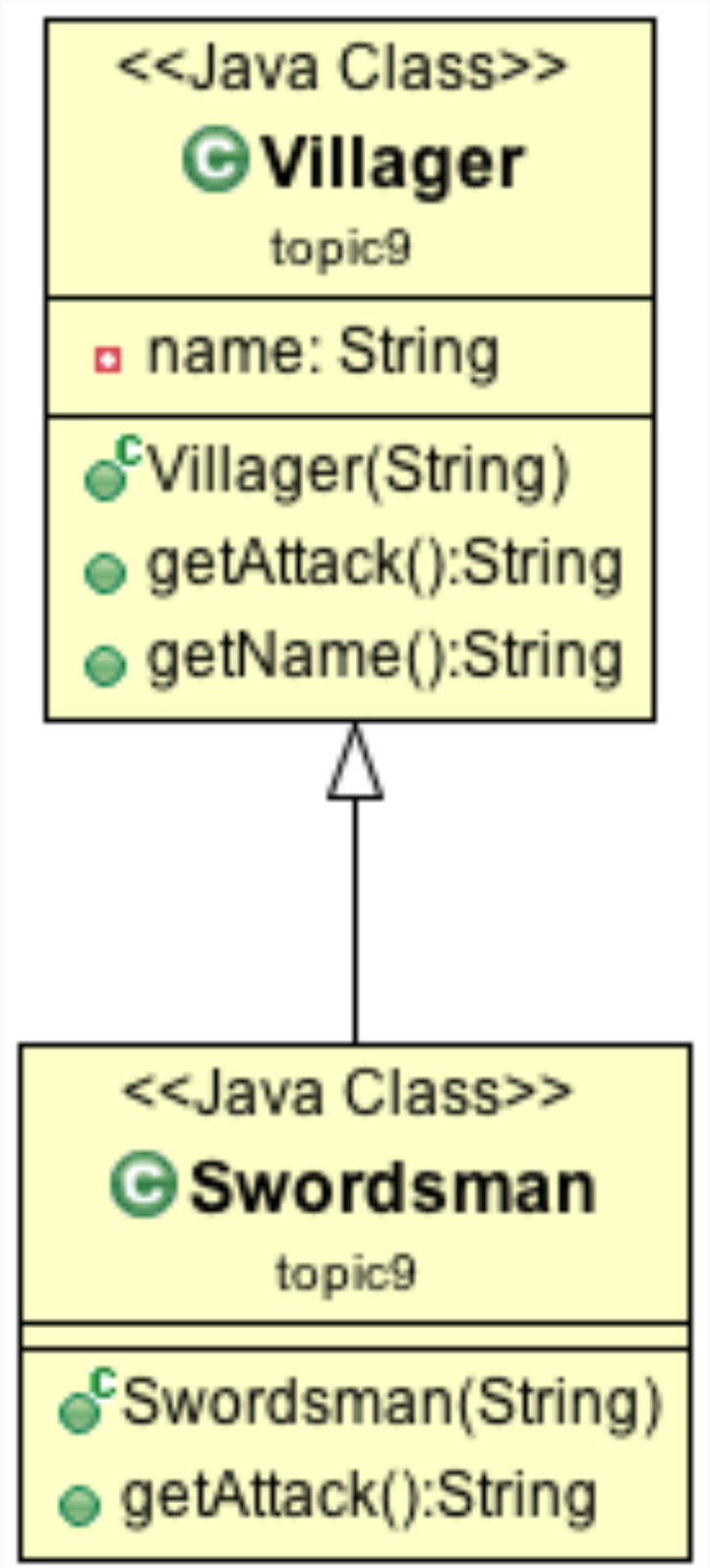
```
public class Swordsman extends Villager {
```

We use the keywords **extends** to indicate that one class inherits (copies) from another.

- We call the class that does the extending (e.g. `Swordsman`) the **subclass** (or the **child** class) and the class that is extended from the **superclass** (e.g. `Villager`; the **parent** class).

- It is the act of extension that causes the subclass to inherit (copy) everything from the superclass.

INDICATING INHERITANCE IN A CLASS DIAGRAM

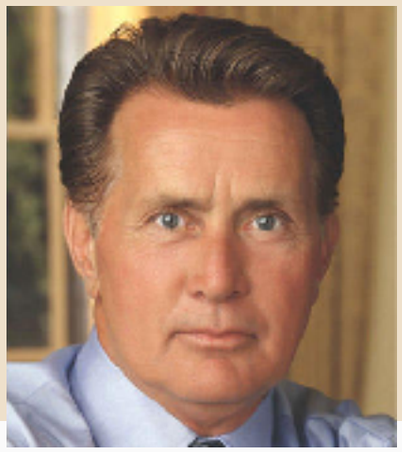


In order to indicate that one class inherits from another in a class diagram, we draw an arrow from the subclass to the superclass, with an **empty arrowhead**.

CONCEPTUAL INHERITANCE

Like most of the concepts in object-oriented programming the concept of inheritance is linked to the **real world**.

```
public class Father {

    public int getActingSkill() {

        return 8;

    }

}
```



```
public class Son extends Father {

    public int getActingSkill() {

        return 3;

    }

}
```

This states that children **are** their parents (which is typically true in terms of physical traits), but also that children tend to **redefine** their parent's behaviours.

OVERRIDING METHODS

```
public String getAttack() {

    return name + " throws a punch";

}
```

Villager.java

```
public String getAttack() {

    return getName() + " swings a sword";

}
```

Swordsman.java

Like the conceptual child redefining the behaviour of a parent, when a subclass extends a superclass, it can **change the behaviour** of the acquired methods by **using the same method signature** (**name** and **parameters**).

- This allows us to **specialise** existing code for new purposes, while also **reusing** other existing code.

- It also **constrains us** to use the **relevant** access modifier and return type (more later).

THE EXTENDS KEYWORD (2): WHAT CAN WE ACCESS (1)

```
public class Villager {

   private String name;

   public Villager(String name) {

      this.name = name;

   }

   public String getAttack() {

      return name + " throws a punch";

   }

   public String getName() {

      return name;

   }

}
```

We've seen that when we extend a class we inherit all the code from that class.

When we inherit from a class, identifiers that are private in the superclass, such as name, **remain private** from outside objects of the subclass, which is intuitive.

THE EXTENDS KEYWORD (2): WHAT CAN WE ACCESS (2)

```
public class Villager {

   private String name;

   public Villager(String name) {

      this.name = name;

   }

   public String getAttack() {

      return name + " throws a punch";

   }

   public String getName() {

      return name;

   }

}
```

Slightly less intuitively, identifiers that are private in the superclass, also **cannot be accessed from the subclass**.

- Parents keep things private from their children.
- Therefore, they are effectively **not** inherited.

THE EXTENDS KEYWORD (2): WHAT CAN WE ACCESS (3)

```java
public String getAttack() {

    return getName() + " swings a sword";

}
```

Swordsman.java

Because private fields aren't accessible in a subclass, we cannot reference them directly in the subclass, but we can call (public) methods from the superclass as if they were in the **same class**, as they have been inherited.

- This allows us to access values that have been placed into these **private** fields in the superclass from the subclass, such as the values passed to the superclass constructor, which we look at now:

THE SUPER KEYWORD (PART 1) (1)

```
public Villager(String name) {

   this.name = name;

}
```

```
public Swordsman(String name) {

   super(name);

}
```

Among the identifiers inherited from a superclass is that superclass's **constructor**(s).

- To construct a subclass, you must **first** call **at least one** of the constructor(s) of the superclass.
    - We shouldn't lose the **restrictions** put in place by the superclass, as they likely relate to inherited identifiers. For example, here, `Villager` inherits `getName`, which relies on a value being present in the `name` field.
- This is done by the subclass's constructor.

REMEMBER: LECTURE EXERCISE: BOOK (9), ADDITIONAL CONSTRUCTOR

```
    private int currentPage;
    private int totalPages;

    public Book(int totalPages) {

        currentPage = 1;
        this.totalPages = totalPages;

    }

    public Book() {

        this(0);

    }
```

Remember multiple constructors give us different patterns we can match when we create an object of a class.

THE SUPER KEYWORD (PART 1) (2)

```
public Villager(String name) {

   this.name = name;

}
```

```
public Swordsman(String name) {

   super(name);

}
```

- To differentiate between the identifiers that exist in the subclass, and those that **come from** the superclass we use the keyword **super**.
    - Like writing **this** (with parameters) to call a constructor in the same class, we simply write **super** (with parameters) to call a constructor in the superclass.
- If the superclass has an empty or default constructor, a super call will be added **implicitly.**
    - This **sets up** the parent class.

ACCESSING ORIGINAL METHODS (THE SUPER KEYWORD (PART 2))

Sometimes it might be useful to access the information returned from a method **before** it is overridden.

As with the constructor, we use the keyword **super** to differentiate between the method in the superclass, and the method in the subclass.

- However this time we use it almost like a special object, that references the superclass, and after which we place a dot.

```
public class Father {

   public int getActingSk

      return 8;

   }

}
```

```
public class Son extends Father {

   public int getActingSkill() {

      return super.getActingSkill() / 2.0;

   }

}
```

We might imagine the a son is only ever half as good an actor as their father.

ASIDE: OVERRIDING FIELDS?

We can't override fields from the parent class in the same way.

- If they are **private**, they aren't accessible anyway.

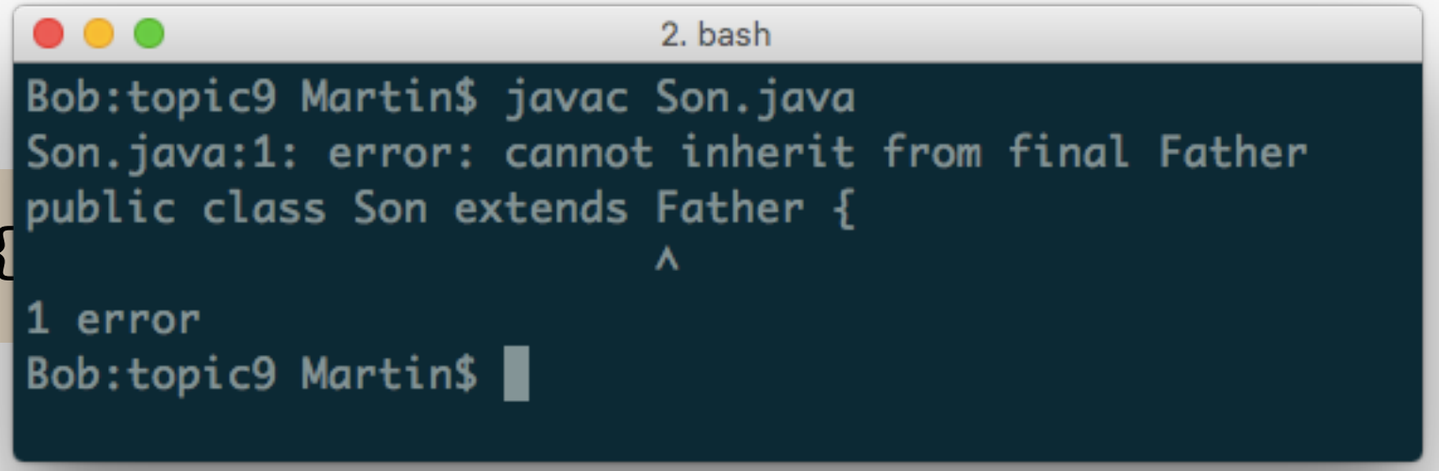
- If they happened to be **public**, they will be accessible, and fields of the same name in the subclass will **hide** the field inherited from the superclass.

- However, as public fields should also be final and static, they should be accessed directly through the class anyway, negating this issue.

- If they must be accessed through the instance, then the super keyword can be used to differentiate, as it can with methods.

ASIDE: THE IMPACT OF FINAL ON INHERITANCE

```
public final class Father {
```



It hopefully makes intuitive sense that **final** classes cannot be extended.

Therefore, it also hopefully makes intuitive sense that final methods cannot be overridden.

We might see this as yet another way that object-orientation allows us to **control** how our code is used.

- We saw other ways in which object-orientation allows us to control how our code is used in Topic 4.

LECTURE EXERCISE: EXAMS AND MARK SCHEMES

```
public class BooleanQuestion {

    private boolean answer;
    private int mark;

    public BooleanQuestion(boolean answer, int mark) {

        this.answer = answer;
        this.mark = mark;

    }

    public boolean lookAtAnswer() {

        return answer;

    }
```

```
public class NumericalQuestion {

    private int answer;
    private int mark;

    public NumericalQuestion(int answer, int mark) {

        this.answer = answer;
        this.mark = mark;

    }

    public int lookAtAnswer() {

        return answer;

    }
```

The next few slides are tough, stay with me...

POLYMORPHISM (1)

Because a subclass is the same **type** as its superclass, we can legally do the following:

```
Villager swordsman = new Swordsman("Swordsman");
```

This provides us with important flexibility when representing a set of villagers in our village:

BACK TO THE MEDIEVAL VILLAGE (FIRST PASS)

```
public class Village {

   public static void main(String[] args) {

      ArrayList<Villager> villagers = new ArrayList<Villager>();
      System.out.println("Who is in the village?");

      for ( int i = 0; i < Integer.parseInt(args[0]); i++ ) {

         int selection = ((int)(Math.random() * 2));

      if ( selection == 0 ) {

         villagers.add(new Villager("Villager"));

      } else {

         villagers.add(new Swordsman("Swordsman"));

      }

   }
```

Based on a command line argument (N), create N villagers of a random type.

```
ArrayList<Villager> villagers = new ArrayList<Villager>();
villagers.add(new Swordsman("Swordsman"));
```

We've seen that both arrays and arraylists can only store values of the **same type**.

It might therefore seem strange that I can add villagers **and** swordsmen to an arraylist that is typed as a `Villager`.

But recall that this homogenous restriction is in place in the first place to ensure that all elements in the list **act** in the same way.

Because `Swordsman` and `Villager` now share a common interface via inheritance (a swordsman IS-A villager), a swordsman will behave in **at least** the same way as a villager, so can be stored in the **same list**.

POLYMORPHISM (2)

```
Villager swordsman = new Swordsman("Swordsman");
```

As well as the above, the following is still possible:

```
Villager villager = new Villager("Villager");
```

This means we could end up in a situation in which in the same program, **two variables of the same type** (the same interface; `Villager`) present **different behaviour**.

POLYMORPHISM (3)

```
Villager villager = new Villager("Villager");

Villager swordsman = new Swordsman("Swordsman");
```

Same interface

Different form

This is known as **polymorphism.**

- Sometimes a `Villager` takes the form of itself, sometimes of its child class `Swordsman` (it has many (**poly**) forms (**morph**)).

- Because method overriding **facilitates** this difference in behaviour, the two terms are often used **interchangeably**.

This flexibility is useful for our fight:

THE VILLAGE (FIRST PASS): POLYMORPHISM (AND FIGHTING TROLLS) (1)

Continued in the main method of Village:

```
System.out.println("A troll attacks the village");

for ( Villager villager : villagers ) {

    System.out.println(villager.getAttack());

}
```

Village.java

We can iterate through this list, and call **any method** in the **parent type**.

- If it has been overridden, we will call the overridden method **instead**.
- So our villagers fight in **their own ways**, as desired.

THE VILLAGE (FIRST PASS): POLYMORPHISM (AND FIGHTING TROLLS) (2)

Continued in the main method of Village:

```
System.out.println("A troll attacks the village");

for ( Villager villager : villagers ) {

    System.out.println(villager.getAttack());

}
```

Village.java

We can iterate through this list, and call **any method** in the **parent type**.

- We are still interacting with the **same** `Villager` **interface each time** but the behaviour behind that interface is different depending on the **actual type** of each object in the arraylist.

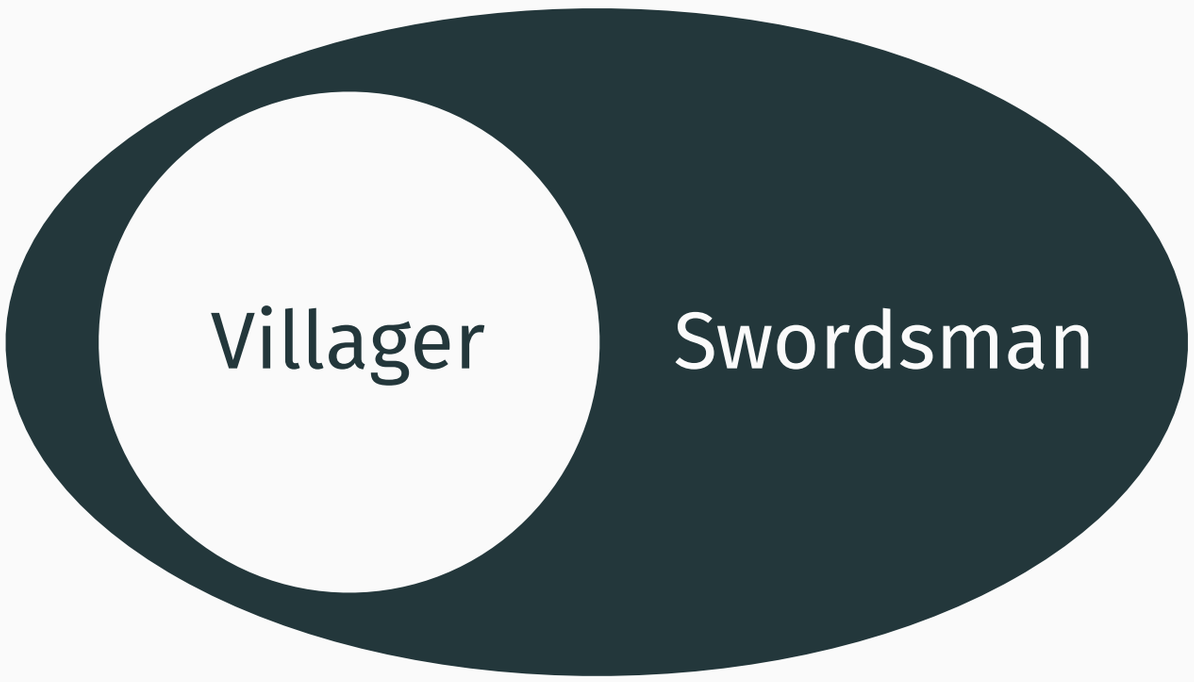
- But we still **reach** the new behaviour **through** the `Villager` interface. This has implications (more later).

ASIDE: SUPERCLASS AS A SUBCLASS?

```
Swordsman anotherSwordsman = new Villager("Another Swordsman");
```

Can a `Villager` also be a `Swordsman`?

No. Because we can **add to the interface of the subclass without changing the interface of the superclass**, we cannot guarantee that the **entire interface** of `Swordsman` will be **supported** by the `Villager` object.



`Villagers` and `Swordsman` share a common interface, but this doesn't mean that the Swordsman cannot add to this interface (**more shortly**).

POLYMORPHISM (METHODS AND RETURN TYPES) (1)

We can tighten up our representation of a village:

```
public class Village {

   private ArrayList<Villager> villagers;

   public Village() {

      villagers = new ArrayList<Villager>();

   }

   public void addVillager(Villager villager) {

      villagers.add(villager);
   }

   public void addRandomVillager() {

      villagers.add(randomVillager());

   }
```

It's not only lists that are **polymorphic**, **parameters** and **return** types can be too.

Any type of Villager can be added to a Village.

POLYMORPHISM (METHODS AND RETURN TYPES) (2)

```
public String fight(Villager villager) {
    return villager.getAttack();
}

public void allFight() {
    for ( Villager villager : villagers ) {
        System.out.println(fight(villager));
    }
}
```

Any villager is made to attack.

Village.java

It's not only lists that are **polymorphic**, **parameters** and **return** types can be too.

POLYMORPHISM (METHODS AND RETURN TYPES) (3)

```
public Villager randomVillager() {

    int selection = ((int)(Math.random() * 2));

    if ( selection == 0 ) {

        return new Villager("Villager");

    } else {

        return new Swordsman("Swordsman");

    }

}
```

It's not only lists that are **polymorphic**, **parameters** and **return** types can be too.

Any type of villager can be returned from this method.

Village.java

TROLL FIGHT (PART 1)

```
public class TrollFight {

    public static void main(String[] args) {

        Village village = new Village();

        for ( int i = 0; i < Integer.parseInt(args[0]); i++ ) {

            village.addRandomVillager();

        }

        System.out.println("A troll attacks the village");

        village.allFight();

    }

}
```

We'll return to our troll fight later.

POLYMORPHISM (REUSABILITY)

This flexibility offer by polymorphism makes our programs very easy to **reuse** and **extend**.

- Someone else can make a Village object, and create a new type of villager (e.g a bowman) that can then added to the village, and made to fight, without further modification.

This is an idea we can also see more practically in the **Java library**, along with the notions of inheritance and overriding.

Inheritance And The Java Library

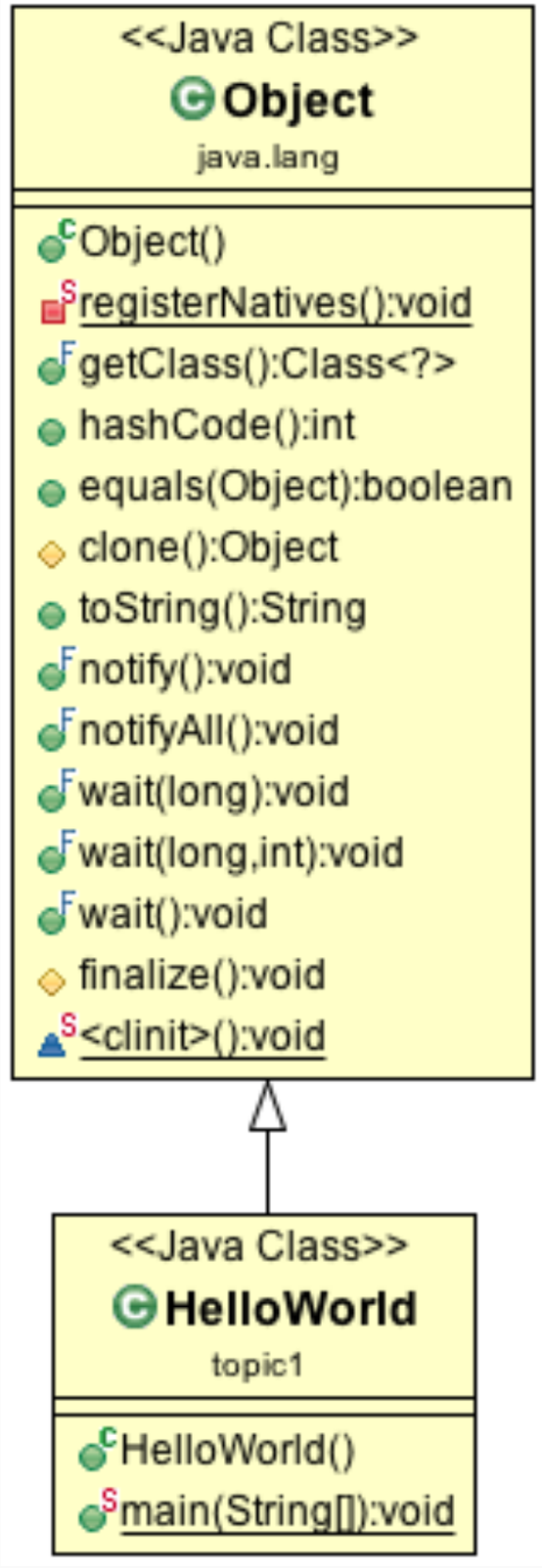
OBJECT

We've already been using inheritance, and indeed overriding methods, in our programs (probably) without realising it.

All classes implicitly inherit from a Java library class called **Object**.

- As all subclasses must call at least one constructor of their parent class, **implicit super calls** are added to each of our classes to the constructor of Object.

We should see some methods here that are **familiar** to us.



REMEMBER: THE TOSTRING() METHOD (2)

In the coordinates class:

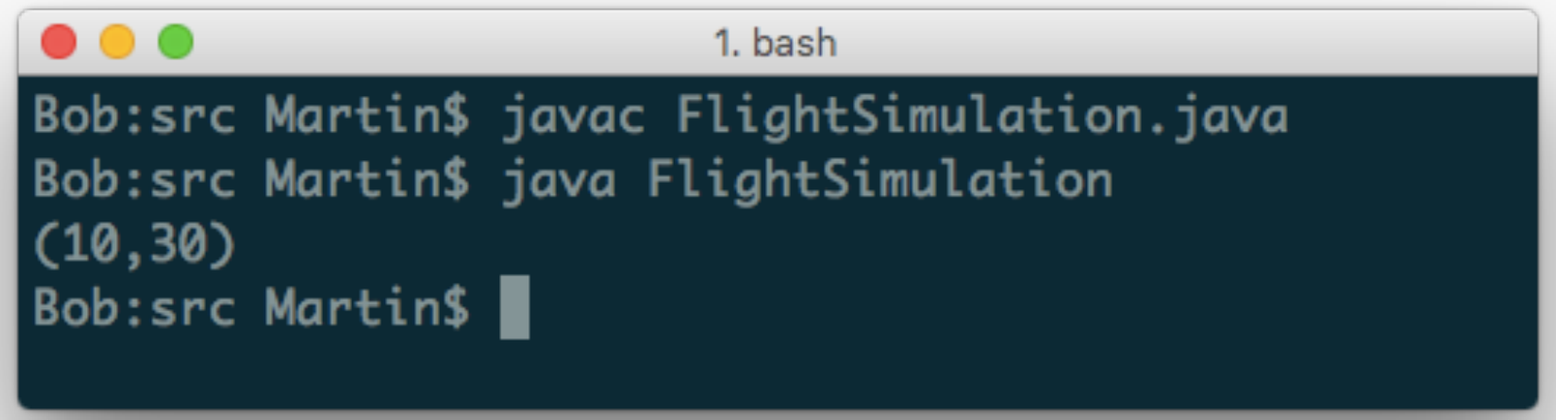
You **must** return a String.

You **cannot** supply any parameters.

```
public String toString() {

    return "(" + x + "," + y + ")";

}
```

In a driving class:

```
Coordinates planeCoordinates = new Coordinates(10, 30);
System.out.println(planeCoordinates);
```



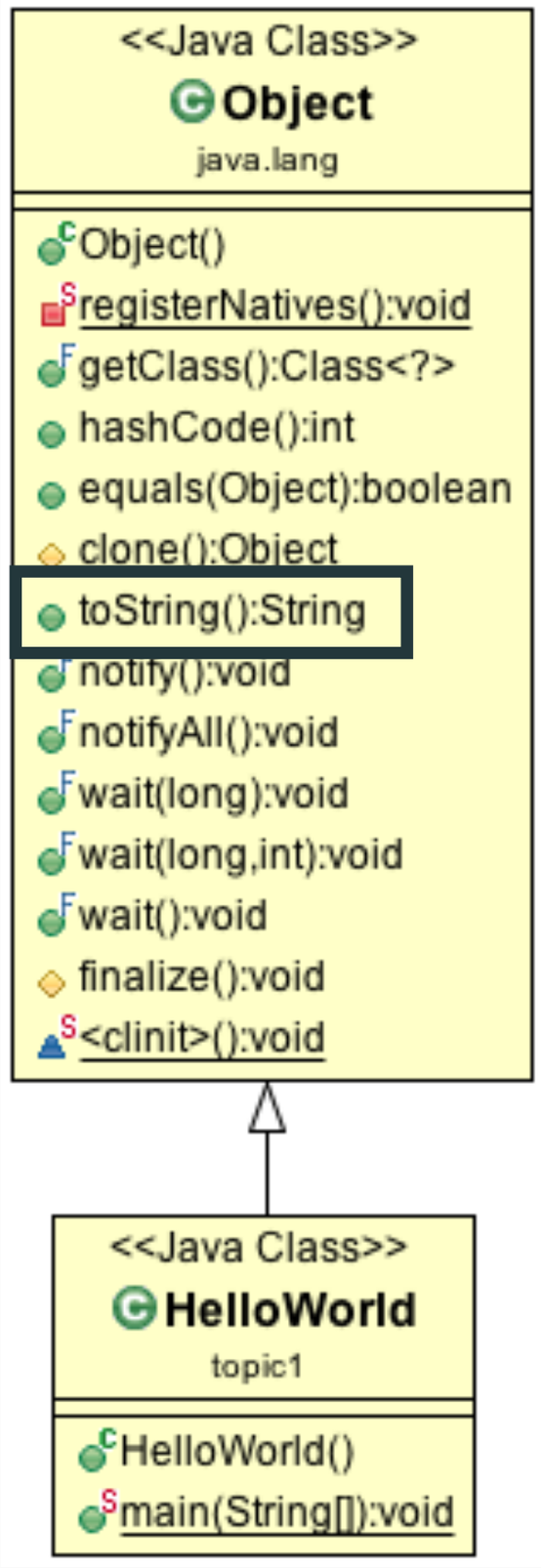
TOSTRING

The most familiar of the methods in the Object class is (probably) `toString`.

Now it should be clearer why the JVM looks for a `toString` method in order to print a string representation of an object: because it knows that **all objects** have (**inherit**) this method.

When we write our own `toString` method we **override** the default functionality (by keeping the same name and parameters, and not altering the return type), such that the JVM still looks for the `toString` method, but instead now uses **our implementation**.

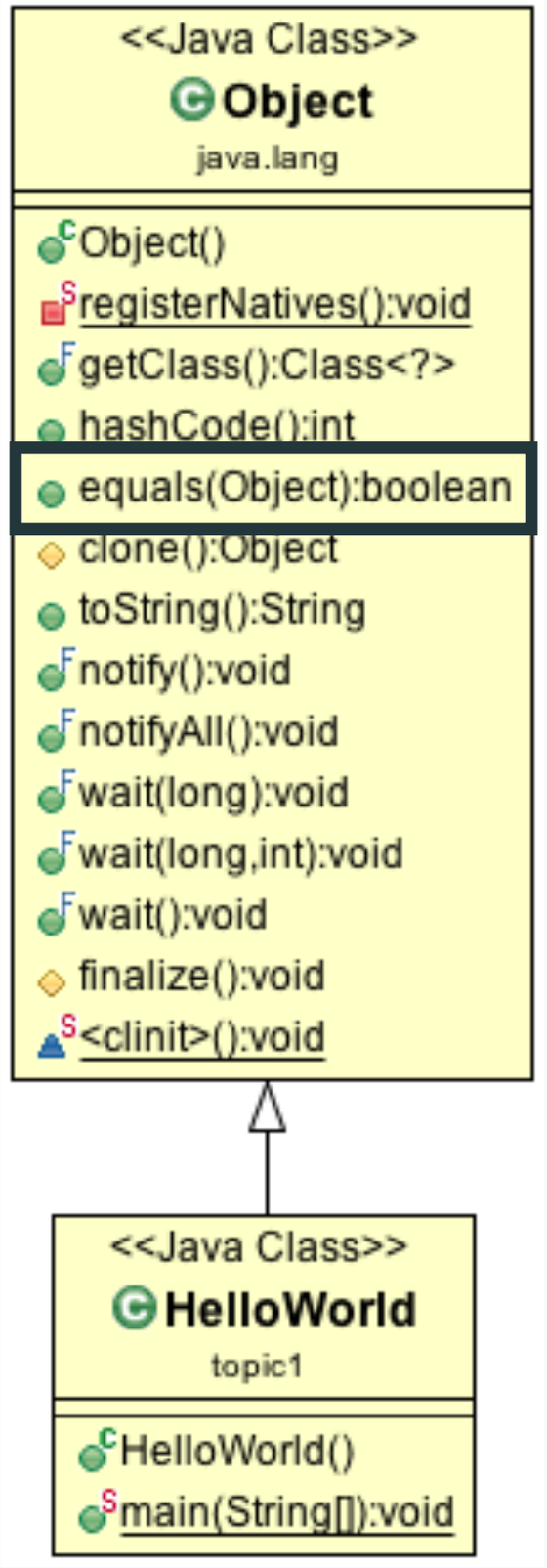
- The **default implementation** of `toString` inside `Object` is to return the name of the current class, and a memory location.



EQUALS

Another familiar method here should be `equals`.

Let’s look at where we’ve seen this used before, and what we can now understand about its use given our knowledge of inheritance and overriding.



REMEMBER: BACK TO COPYING STRINGS: ASKING ABOUT COPIES

```
public static void main(String[] args) {

      String a = "A";

      String b = "A";

      String c = "A";

      String d = "A";

      System.out.println(a == b);

}
```

We ask the JVM, `are you sharing the same string copy between a and b'?

*Are a and b **equivalent in memory**?*

*This answer one of our open questions about the **flexibility** of relational operators.*

In order to confirm that the JVM is sharing copies between our variables, we can use the familiar **comparison** operator.

- This will print **true**.

REMEMBER: COMPARING STRINGS (4)

We use the **.equals** method in the String class* to compare strings.

**it actually exists elsewhere, more to come.*

```
public static void main(String[] args) {

    String martin = "martin";
    String mart = "mart";
    String in = "in";

    System.out.println(mart + in);
    System.out.println(martin);
    System.out.println(martin == mart + in);
    System.out.println(martin.equals(mart + in));

}
```

COMPARING OBJECTS (1)

Because Strings are **objects**, then the way in which objects are compared more generally should be similar to the way in which we compare Strings.

- We don't usually want to rely on **object identity** (==) to determine if two objects are equal, even though in some cases this may return a correct result.

- Like Strings, two objects may have **similar values in their fields**, and thus be considered equal, but be held **separately** in memory.

- Instead, if we want to compare objects of our own classes, we should **override the equals method** (which is exactly what the `String` class does) to do this.

  - The default behaviour of equals is to simply compare object identity (==).

EXAMPLE: COMPARING COORDINATES (1)

```
Coordinates destination1Coordinates = new Coordinates(110, 135);
Destination destination1 = new Destination("Buenos Aires",
                                           destination1Coordinates);

Aeroplane plane = new Aeroplane("Spirit", destination1Coordinates,
                                                    12, 0, 2550);
```

Prints true.

```
System.out.println(destination1.getCoordinates() ==
                                           plane.getCoordinates());
```

```
Coordinates destination1Coordinates = new Coordinates(110, 135);
Destination destination1 = new Destination("Buenos Aires",
                                           destination1Coordinates);

Coordinates planeCoordinates = new Coordinates(110, 135);
Aeroplane plane = new Aeroplane("Spirit", planeCoordinates,
                                                    12, 0, 2550);
```

Should really print true.

```
System.out.println(destination1.getCoordinates() ==
                                           plane.getCoordinates());
```

EXAMPLE: COMPARING COORDINATES (2): IMPLEMENTING EQUALS (1)

```
public boolean equals(Object coordinates) {



}
```

Like `toString`, we use the same method name and parameters, while keeping the return type consistent to override the equals method in our `Coordinates` class.

Here though, we see the **limitations of specialisation**: we **can't** change the signature, otherwise the method **won't** be overridden, and thus `equals`, which accepts an object as a parameter in the `Object` class, can only ever accept an object.

What's the solution?

TROLL FIGHT (PART 2)

Let’s answer this by looking at a more general problem in our troll fight.

```
public class TrollFight {

    public static void main(String[] args) {

        Village village = new Village();

        for ( int i = 0; i < Integer.parseInt(args[0]); i++ ) {
            village.addRandomVillager();
        }

        System.out.println("A troll attacks the village");

        village.allFight();
    }
}
```

BACK TO THE MEDIEVAL VILLAGE: MAGES

MAGES: IN CODE

```
public class Mage extends Villager {

    private int damage;

    public Mage(String name, int damage) {

        super(name);
        this.damage = damage;

    }

    public String getAttack() {

        return getName() + "
              casts a spell with damage " + damage;

    }

    public int getDamage() {

        return damage;

    }

}
```

Adding **new fields** and **new methods** is also part of the specialisation process.

If we wish to do more work in a subclass constructor (e.g. assigning new fields), the **call to the super constructor must come first**.

POLYMORPHISM (METHODS AND RETURN TYPES) (2)

```
public Villager randomVillager() {

    int selection = ((int)(Math.random() * 3));

    if ( selection == 0 ) {

        return new Villager("Villager");

    } else if ( selection == 1 ){

        return new Swordsman("Swordsman");

    } else {

        return new Mage("Mage",
                        ((int)(Math.random() * 10) + 1));

    }

}
```

We can update our randomVillager *method accordingly to account for the presence of a mage.*

Mages are given random damage between 1 and 10.

Village.java

TROLL FIGHT (PART 2): WHAT CAN KILL A TROLL?

```
public boolean allFight() {

   int totalDamage = 0;

   for ( Villager villager : villagers ) {

      System.out.println(fight(villager));
      totalDamage += villager.getDamage();

   }

   if ( totalDamage > 50) {

      return true;

   }

   return false;

}
```

*There's a problem: this code will **not** compile.*

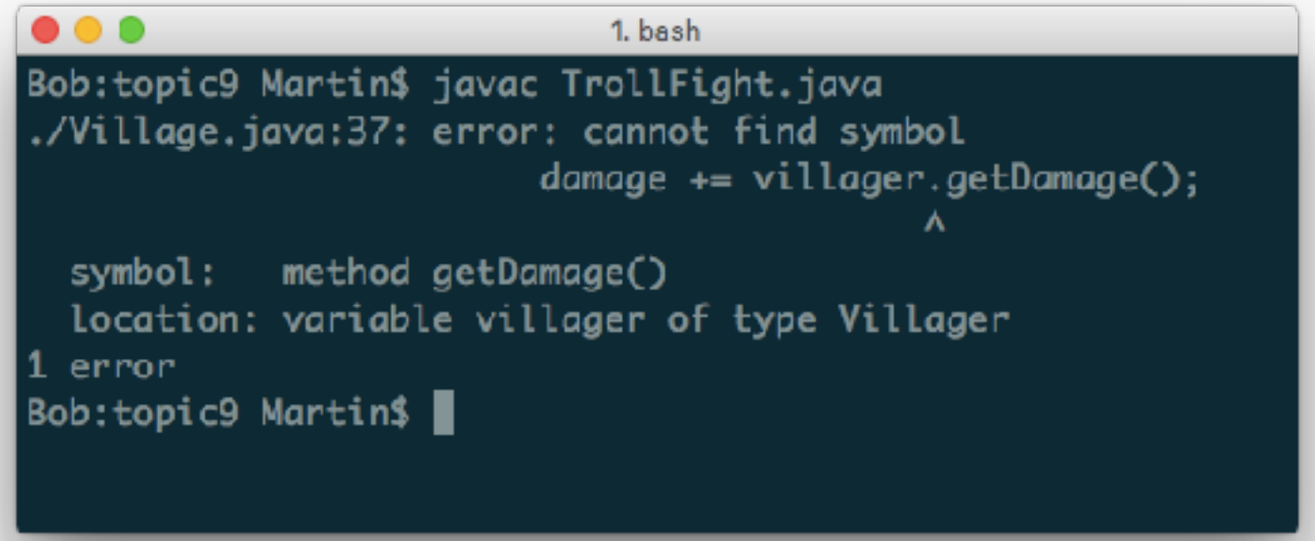
Village.java

Remember that villagers and swordsmen have little effect in a fight (against a troll), which we represent using simple print statements.

Mages, however, have damage, which does affect the troll.

We can represent this by collecting the **total damage** done by all the mages in our party (remember we select villagers at random) and seeing if it is enough to defeat the troll (50).

ERROR: CANNOT FIND SYMBOL



This makes sense, because `getDamage` is a method unique to a mage, not to a villager.

```
for ( Villager villager : villagers ) {

   ...
   totalDamage += villager.getDamage();

}
```

Remember that although the behaviour specified by each object in the villagers list may be different, we are still only interacting with the `Villager` interface, so anything that **isn't** in that `Villager` interface **cannot** be referenced.

TROLL FIGHT (PART 2): WHAT CAN KILL A TROLL?

```
public boolean allFight() {

   int totalDamage = 0;

   for ( Villager villager : villagers ) {

      System.out.println(fight(villager));
      totalDamage +=
           ((Mage)villager).getDamage();
   }

   if ( totalDamage > 50) {

      return true;

   }

   return false;
}
```

Village.java

The solution is to **temporarily force** a villager to **behave** as a mage, so that we can interact with the Mage interface, **rather** than the `Villager` interface.

We want to **cast** a `Villager` type as a Mage type, which works because (**some**) villagers are also mages.

REMEMBER: ASIDE: CASTING

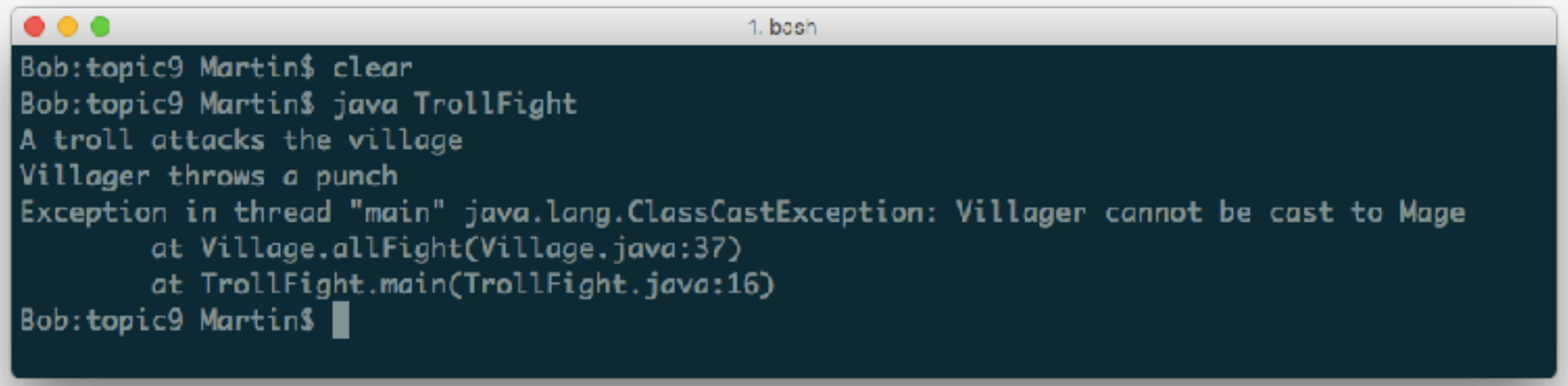
Why `casting'? Because the **same value** is **cast in different roles**.

THE DANGERS OF CASTING

There's a problem here though. Not all of our villagers are mages. What happens if we try and cast an object that isn't of the Mage type as a mage?

```
Mage mage = ((Mage)swordsman);
```

We get an **error**. Therefore, we can be fairly sure that if we try and perform this cast at **runtime** (i.e. when the compiler cannot check exactly what is in the villagers array), we will also get an error:



INSTANCE OF

We need a way to look **past** the `Villager` interface and test the **actual type** of the object in the array to see if it is **of the same type that we wish to cast to**.

We can do this using a new operator called **instance of**:

```
for ( Villager villager : villagers ) {
   System.out.println(fight(villager));

   if ( villager instanceof Mage ) {

      damage += ((Mage)villager).getDamage();

   }

}
```

Village.java

*This operator takes an object as its first operand and a **type** as its second.*

EXAMPLE: COMPARING COORDINATES (2): IMPLEMENTING EQUALS (1)

```
public boolean equals(Object coordinates) {

   if ( !(coordinates instanceof Coordinates) ) return false;

   return x == ((Coordinates)coordinates).x && y ==
                                    ((Coordinates)coordinates).y;

}
```

We can now apply the same thinking to our equals method for coordinates.

We **cast** the parameter of type `Object`, which works because we can cast a super type to a subtype, as we have seen, and we do so safely using instance of.

So our functionality is as expected:

EXAMPLE: COMPARING COORDINATES (2): IMPLEMENTING EQUALS (2)

```
Coordinates destination1Coordinates = new Coordinates(110, 135);
Destination destination1 = new Destination("Buenos Aires",
                                           destination1Coordinates);

Aeroplane plane = new Aeroplane("Spirit", destination1Coordinates,
                                                    12, 0, 2550);
```

Prints true.

```
System.out.println(destination1.getCoordinates().equals(
                                           plane.getCoordinates()));
```

```
Coordinates destination1Coordinates = new Coordinates(110, 135);
Destination destination1 = new Destination("Buenos Aires",
                                           destination1Coordinates);

Coordinates planeCoordinates = new Coordinates(110, 135);
Aeroplane plane = new Aeroplane("Spirit", planeCoordinates,
                                                    12, 0, 2550);
```

Prints true.

```
System.out.println(destination1.getCoordinates().equals(
                                           plane.getCoordinates()));
```

The next few slides are also tough, stay with me...

SERVICE PROVIDERS (1)

A valid question here is, why override equals? Why not simply compare objects using a method with a name of our **own choosing** or just using accessor methods?

- In our Noughts and Crosses game, we simply used a method called `matches`.

Partly, it's **convention** to use .equals to compare objects.

More subtly, remember the notion of **polymorphic arguments**.

- In our `Villager` class, we saw that no matter which **type** of `Villager` (one that already exists, or a new type) is passed to the `fight` method, they can have the method `getAttack` invoked on them.
  - This is because all villagers have this method, overridden or not.
- We might think of `fight` as providing a **reusable service** — prompting a villager to fight (against a troll) — **using a method provided** in the Villager class (`getAttack`).

REMEMBER: POLYMORPHISM (METHODS AND RETURN TYPES) (2)

```
public String fight(Villager villager) {

   return villager.getAttack();
}

public void allFight() {

   for ( Villager villager : villagers ) {

      System.out.println(fight(villager));

   }
}
```

Any villager is made to attack.

It's not only lists that are **polymorphic**, **parameters** and **return** types can be too.

SERVICE PROVIDERS (2)

To use this service (`fight`), we must pass an object that is capable of **supporting** the **task carried out by the service**; an object that has a `getAttack` method; a villager that can fight.

- In other words we can only pass villagers, or their descendent classes.

- This makes the service **exclusive**. We will look at this idea in the final section of the topic.

This service only works in an **expected way** if the `getAttack` method is **overridden correctly**.

- Otherwise the method from the superclass (Villager) will be called **instead**.

Moreover, this service only works in an expected way if something **sensible** is returned from an overridden `getAttack` method.

- If, for example, the string `Run Away' were returned, then the output would not be as expected.

SERVICE PROVIDERS (3)

Many methods in library classes operate in the same way

- They accept a parameter of type `Object`.
- This means that no restrictions are placed on the extension hierarchy of the objects passed to these methods (the service is **non-exclusive**).
- Methods from the `Object` class can still be called on any passed objects in order to **achieve their advertised service**, because these methods exist in every object.
- If we want to use this service, and have it behave in a way that we **expect**, we have to pass objects of our own classes to it with methods overridden in a **certain way**.

CONTAINS

contains

`public boolean contains(Object o)`

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Specified by:

`contains` in interface `Collection<E>`

Specified by:

`contains` in interface `List<E>`

Overrides:

`contains` in class `AbstractCollection<E>`

Parameters:

`o` - element whose presence in this list is to be tested

Returns:

`true` if this list contains the specified element

The `contains` method in the `ArrayList` class operates in this way, as a service provider.

The service it offers is to return you **true** if the ArrayList contains the supplied object.

CONTAINS

contains

`public boolean contains(Object o)`

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Specified by:

`contains` in interface `Collection<E>`

Specified by:

`contains` in interface `List<E>`

Overrides:

`contains` in class `AbstractCollection<E>`

Parameters:

`o` - element whose presence in this list is to be tested

Returns:

`true` if this list contains the specified element

To **perform** this service, the `contains` method relies on the existence of the equals method (which is confirmed).

But for the service to work **as expected** (i.e. for an object to be correctly located in the list) the equals method must be **overridden correctly** and **implemented in a manner that allows for the correct expression of object equality**.

CONTAINS EXAMPLE: FINDING AN EQUAL COORDINATE (1)

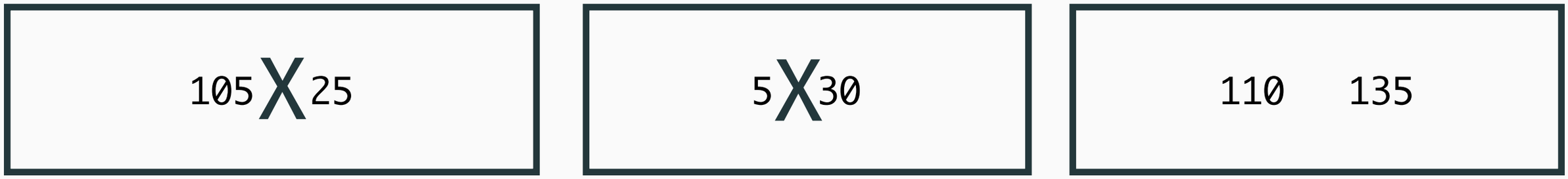
Let's understand how contains works for a list of coordinates, and thus why overriding `equals` in our `Coordinates` object in the correct way allows `contains` to successfully provide its advertised service:

```
Coordinates destination1Coordinates = new Coordinates(110, 135);
Destination destination1 = new Destination("Buenos Aires", destination1Coordinates);

Coordinates destination2Coordinates = new Coordinates(5, 30);
Destination destination2 = new Destination("Muscat", destination2Coordinates);

Coordinates destination3Coordinates = new Coordinates(105, 25);
Destination destination3 = new Destination("Zanzibar", destination3Coordinates);

ArrayList<Coordinates> coordinates = new ArrayList<Coordinates>();
coordinates.add(destination3Coordinates);
coordinates.add(destination2Coordinates);
coordinates.add(destination1Coordinates);

Coordinates aeroplaneCoordinates = new Coordinates(110, 135);

System.out.println(coordinates.contains(aeroplaneCoordinates));
```

CONTAINS EXAMPLE: FINDING AN EQUAL COORDINATE (2)

The contains method repeatedly calls `equals` on its object argument, and each time passes each element of the list as an argument. If `equals` returns true at any point, `contains` returns true:

`ArrayList<Coordinates> coordinates`




```
Coordinates aeroplaneCoordinates =
                        new Coordinates(110, 135);
```

```
public boolean equals(Object coordinates) {

   if ( !(coordinates instanceof Coordinates) ) return false;
                                                    135
   return x == ((Coordinates)coordinates).x && y ==
         110                              ((Coordinates)coordinates).y;

}
```

CONTAINS EXAMPLE: FINDING AN EQUAL COORDINATE (2)

Without an adequately implemented equals method, or without overriding equals properly, contains cannot fulfil its advertised service.

ArrayList<Coordinates> coordinates

105 25	5 30	110 135

```
Coordinates aeroplaneCoordinates =
                    new Coordinates(110, 135);
```

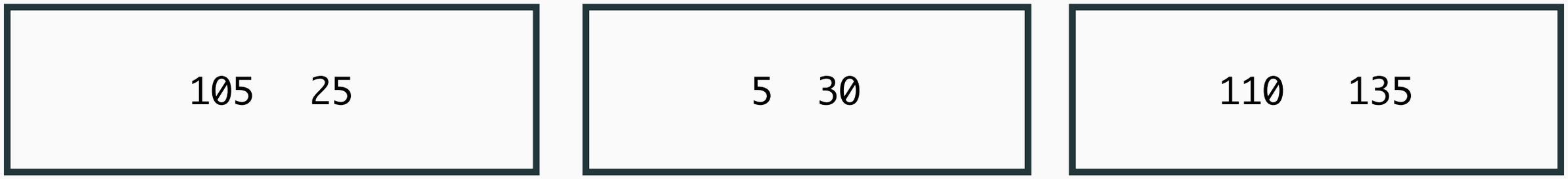
```
public boolean equals(Object coordinates) {

    return true;

}
```

If we just returned true, contains would assume that anything was contained in our list

CONTAINS EXAMPLE: FINDING AN EQUAL COORDINATE (2)

Without an adequately implemented equals method, or without overriding equals properly, contains cannot fulfil its advertised service.

ArrayList<Coordinates> coordinates




```
Coordinates aeroplaneCoordinates =
                    new Coordinates(110, 135);
```

```
public boolean equals(Coordinates coordinates) {
    // If we change the parameter type to Coordinate, we
    // aren't really supplying a true equals method: it's
    // just a method that happens to be called equals
    // (not the same signature), and contains uses the
    // default implementation of equals from Object.
}
```

ASIDE: SAFELY OVERRIDING EQUALS

Most of the time equals will work as expected under a simple implementation, but you may wish to check the following in order to ensure that equals **always** works (i.e. these are **guidelines**):

- Ensure equals is **symmetric** ("If I compare CoordinateA to CoordinateB, and return true, do I also return true if I compare CoordinateB to CoordinateA?")

- Ensure equals is **reflexive** ("If I compare CoordinateA to CoordinateA, do I get a true value?").

- Implement a supplementary **Hashcode** method, but typically only required if you intend to store your objects in **hashed** data structures, which we have not looked at.

    - Strict **warnings** about this from the compiler can thus be ignored (like the lack of a close method when using Scanner).

ASIDE: ACCESSIBLE IDENTIFIERS (1)

Recall that, currently, the only **accessible identifiers** that are inherited from a class are those marked **public.**

- **Private** identifiers remain private to the parent.

There is another modifier available to us that makes an identifier accessible:

```
public class Villager {

    protected String name;
```

Protected identifiers are **private to the class, and all subclasses** (technically).

In **reality**, given what we've seen **so far**, there's no difference between public, protected and fields without an access modifier (more next semester).

ASIDE: ACCESSIBLE IDENTIFIERS (2)

Taking this technical definition of protected, I still tend to rely on accessors in the subclass, and keep things private.

- If we send our class out into the world and someone **extends** it, they could compromise encapsulation by accessing protected fields directly from a subclass.

I'm more likely to use **protected methods**.

- Related aside: we haven't looked at **private** methods directly, but hopefully the result of using this access modifier on a method is clear: that method can only be accessed with the class itself.
    - Private methods therefore **support** the functionality offered by other methods.

REMEMBER: ASIDE: RELINQUISHING CONTROL FOR FLEXIBILITY (2)

If we want to do this we can specify **multiple constructors**, each of which accepts different parameters, thus giving a user the option to construct an object of our class in **different ways**.

Each of our constructors has a different pattern, and thus there are different ways to match and create objects of the class.

```
private double balance;

   public BankAccount() {}

   public BankAccount(double deposit) {

      balance = deposit;

   }
```

*Specifically, we use an **empty** constructor. We can see this as **replacing** the **default** constructor we had previously, to remove any restrictions from the creation of objects of our class.*

Open question: Why is it ok for us to break our `methods with unique names' (Topic 3, Slide 14) rule here?

ASIDE: METHOD OVERLOADING AS A FORM OF POLYMORPHISM (1)

We are familiar with the idea of having **multiple constructors**.

The reason this is possible is because methods are **not just differentiated** by their name, but also their parameters (the whole **signature**), as was alluded to in Topic 3.

This idea extends to any method, such that we can use the same name for two methods, so long as the parameters in those method are different:

```
public void addVillagerToParty(Villager villager) {

   villagers.add(villager);

}

public void addVillagerToParty() {

   villagers.add(randomVillager());

}
```

This neatens things up if methods perform broadly the same functionality, except rely on different input.

ASIDE: METHOD OVERLOADING AS A FORM OF POLYMORPHISM (1)

Access modifiers and **return type** cannot be used to differentiate a method:

```
public void fight(Villager villager) {

   System.out.println(villager.getAttack());

}

public String fight(Villager villager) {

   return villager.getAttack();

}
```

*This code will **not** compile.*

Because this is another example of (components of) the same interface (i.e. the method name) taking different **forms**, this is also often considered to be a form of polymorphism.

- But we will consider this a **sub definition**.

ASIDE: REDUCING VISIBILITY IN THE INTERFACE (1)

Given that we now know **what defines** a method, it should be even clearer now why overriding works: we **replace** that method's signature.

```java
public String getAttack() {

   return name + " throws a punch";

 }
```

Villager.java

```java
private String getAttack() {

    return getName() + " swings a sword";

 }
```

Swordsman.java

Given that access modifier and return type are not enough to differentiate a method, is it ok if we change them in the subclass (i.e. the overridden method)?

ASIDE: REDUCING VISIBILITY IN THE INTERFACE (2)

No, because polymorphism (the ability to interact with object of a superclass and a subclass via a superclass variable), relies on the **shared interface** (methods with the same name and parameters) **behaving** in the same way.

- **But** we can change an overridden method to return a **subclass** of a class type returned from a superclass method, as the definition of the superclass is suitably general.

```
public Object getAttack() {

    return name + " throws a punch";

}
```

Villager.java

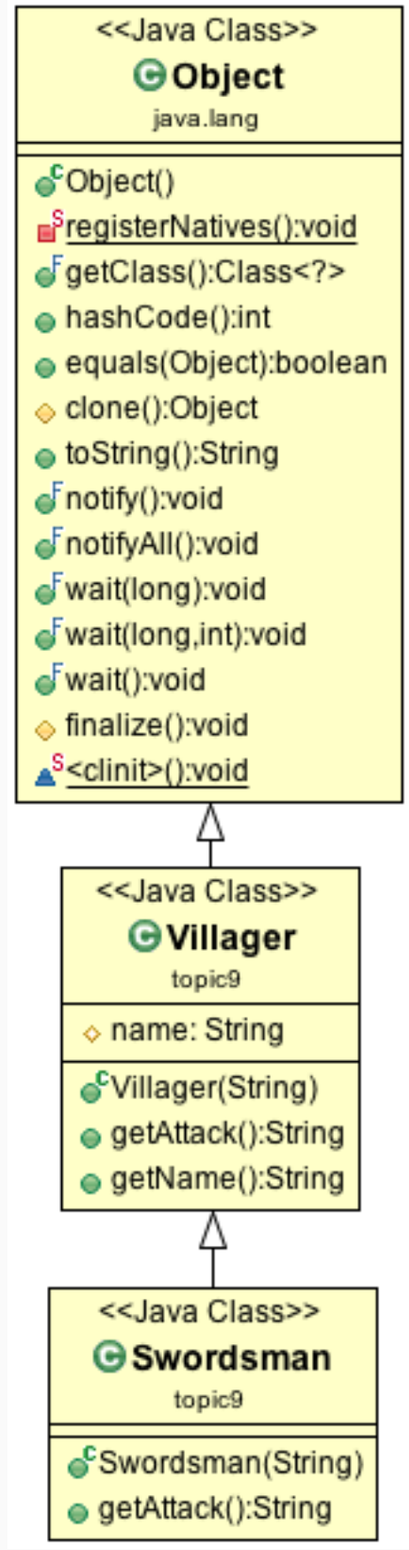
Doing this the other way round would not work, for the reasons described.

```
public String getAttack() {

    return getName() + " swings a sword";

}
```

ASIDE: CLASS HIERARCHY

Given that we now know all classes inherit from `Object`, it should be clear that we can construct hierarchies of classes, in which we inherit from a class that already inherits from another, and so on.

But you can only inherit from **one class directly**. The reasons for this are subtle, and will be presented **next term**.



ASIDE: ARE HAS-A RELATIONSHIPS NOW USELESS? (1)

```
public class Swordsman {

    private Villager villager;

    public Swordsman(String name) {

        this.villager = new Villager(name);

    }

    public String getAttack() {

        return villager.getName() + " swings a sword";

    }

    public String getName() {

        return villager.getName();

    }

}
```

Hopefully it's even more clear now why this HAS-A relationship is awkward.

We wouldn't be able to keep an array of villagers that also contains swordsmen.

ASIDE: ARE HAS-A RELATIONSHIPS NOW USELESS? (2)

Can we extend this idea, such that we just **ditch** the HAS-A relationship and **extend** everything? **Absolutely not**. Would the following make sense?

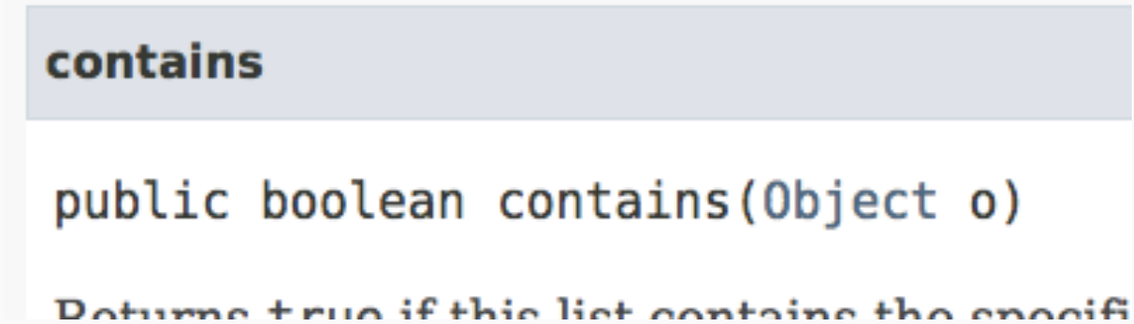
```
public class Aeroplane extends Coordinates {

    private String name;
    private int speed;
    private int totalDistance;
    private int repairDistance;
```

An aeroplane is defined as much more than a coordinate.

This is awkward conceptually, but also tying yourself into a **single class hierarchy**, due to the fact that only one class can be extended, is also limiting, as we will see in the final section of this topic.

Interfaces And Abstract Classes

TYING YOURSELF TO A SINGLE IS-A RELATIONSHIP



Let's think again about the notion of **polymorphic arguments** enabling us to use the **services** of other classes.

- To use a service, so far we've seen that you have to pass an object that extends a certain class.
- This is typically not an issue when class you are expected to extend is `Object` (as this extension is done by default), but in other circumstances it can be.
- Sometimes, to have to **tie one of your classes to being another class just to use a service is limiting**.

BACK TO THE VILLAGE... (1)

BACK TO THE VILLAGE... (2)

Let's consider the introduction of our troll more officially, as an entity that can fight with our villagers:

```
public boolean allFight(Troll troll) {

    int damage = 0;

    for ( Villager villager : villagers ) {

        System.out.println(fight(villager));

        if ( villager instanceof Mage ) {

            damage += ((Mage)villager).getDamage();

        }

    }

    if ( damage >= troll.getHitPoints()) {

        return true;

    }

    return false;

}
```

Village.java

```
public class Troll {

    private int hitPoints;

    public Troll(int hitPoints) {

        this.hitPoints = hitPoints;

    }

    public int getHitPoints() {

        return hitPoints;

    }

}
```

```
Troll troll = new Troll();
village.allFight(troll);
```

TrollFight.java

THE FUN OF A TROLL (1)

In addition to fighting villagers, let's imagine that there are two other ways our troll can interact with a village:

```
private int houses;

public Village() {
    …
    houses = 100;
}

public void intimidate(Troll troll) {
    System.out.println(troll.shout());
}

public void destroy(Troll troll) {
    houses -= troll.getBashRange();
}
```

Village.java

services

```
Troll troll = new Troll(
       "Rawr! Pesky Villagers!", 10, 50);
```

TrollFight.java

```
public class Troll {

    private String phrase;
    private int bashRange, hitPoints;

    public Troll(String phrase … ) {

        this.phrase = phrase;
        …
    }

    public String shout() {

        return phrase;

    }

    public int getBashRange() { … }

    public int getHitPoints() { … }
}
```

BRINGING FRIENDS (1)

Furthermore, let’s imagine that our troll is having so much fun rampaging around the village, that he decides to invite friends: a werewolf and a giant.

BRINGING FRIENDS (2)

Werewolves can intimidate with their howl, but are too small to bash, and giants can bash, but they do not shout.

```
public class Werewolf {

    private String phrase;
    private int hitPoints;

    public Werewolf(String phrase, int hitPoints) {

        this.phrase = phrase;
        this.hitPoints = hitPoints;
    }

    public String shout() {

        return phrase;

    }

    public int getHitPoints() {

        return hitPoints;

    }
}
```

```
public class Giant {

    private int bashRange, hitPoints;

    public Giant(int bashRange, int hitPoints) {

        this.bashRange = bashRange;
        this.hitPoints = hitPoints;
    }

    public int getBashRange() {

        return bashRange;

    }

    public int getHitPoints() {

        return hitPoints;

    }
}
```

FIGHT ANYTHING

Because these two creatures can also fight with the villagers, we could create a **common super class** Enemy, and place hit point information inside this class, such that `Troll`, `Werewolf` and `Giant` all extend this class:

```
public class Enemy {

    private int hitPoints;

    public Enemy(int hitPoints) {

        this.hitPoints = hitPoints;

    }

    public int getHitPoints() {

        return hitPoints;

    }

}
```

```
public class Werewolf extends Enemy {
```

Werewolf.java

```
public class Giant extends Enemy {
```

Giant.java

```
public class Troll extends Enemy {
```

Troll.java

Then any enemy can fight:

```
public boolean allFight(Enemy enemy) {
```

Villager.java

SHOUTING AND BASHING (1)

But what about the other behaviours: `shout` and `bash`?

- We can't put this information in the `Enemy` class, because only giants and trolls can bash, and only werewolves and trolls can shout.
- We can't only put `bash` in `Giant` and `shout` in `Werewolf`, because `Troll` also needs this, and we don't want **repeated** code.
- We can't leave everything in `Troll`, because then giants and werewolves won't have these behaviours.

SHOUTING AND BASHING (2)

A **naïve** choice might be to create two new classes, Shouter and Basher:

```
public class Shouter {

    private String phrase;

    public Shouter(String phrase) {

        this.phrase = phrase;

    }

    public String shout() {

        return phrase;

    }

}
```

```
public class Basher {

    private int bashRange;

    public Basher(int bashRange) {

        this.bashRange = bashRange;

    }

    public int getBashRange() {

        return bashRange;

    }

}
```

SHOUTING AND BASHING (3)

We could then **change the arguments** to the **action** methods in Village, in order to accept objects of this type:

```
public void intimidate(Shouter shouter) {

    System.out.println(shouter.shout());

}
```

Village.java

```
public void destroy(Basher basher) {

    houses -= basher.getBashRange();

}
```

Village.java

And the werewolf and giant could switch up their inheritance to instead be a shouter and a basher, respectively, in order to interact with the village:

```
public class Werewolf extends Shouter {
```

*Services are now **limited** to shouters and bashers.*

```
public class Giant extends Basher {
```

SHOUTING AND BASHING (4)

There are two issues here:

- How do werewolf and giant now **fight** (recall `allFight` accepts an `Enemy`)? `Shouter` and `Basher` could extend `Enemy` (three tiers):

```
public boolean allFight(Enemy enemy) {
```

```
public class Shouter extends Enemy {
```

```
public class Basher extends Enemy {
```

  - But what if villagers want to shout? This would make them enemies too. We have to have a logical **inheritance hierarchy**.

- What about our poor troll?

```
public class Troll extends  ?   {
```

GIVING THE TROLL WHAT HE WANTS

```
public void intimidate(Shouter shouter) {

    System.out.println(shouter.shout());

}
```

Village.java

```
public void destroy(Basher basher) {

    houses -= basher.getBashRange();

}
```

Village.java

Our troll now how to make a decision: do I want to be shouter or a basher (or an enemy)?

- The parameter types required by these methods (services) are **limiting** in that you have to tie a class into a whole **existence** in order for it to use these methods.
- The troll could tie himself to being a shouter, but then he also can't be be a basher (or an enemy) and thus can't destroy, and vice-versa.
- That is, only **one class** can be extended.

But he wants to do everything! What's the solution?

LAYING OUT OUR OPTIONS...

Let's think about our options:

- **Option 1**: Stick to a HAS-A relationship?
- **Option 2**: We need some way to be able to say that a `Troll` IS-A `Shouter` **and** that a `Troll` IS-A `Basher`, so that he can both intimidate and destroy (i.e. use these services), without the **dangers** of multiple inheritance.
- **Option 3**: We want the methods (services) `intimidate` and `destroy` **not** to force our `Troll` (or any of the other creatures) to extend **only** `Shouter` or `Basher` in order to use them, but **still** be **certain** that **the methods they need in order to operate will be available** in any object arguments.

HAS-A RELATIONSHIPS TO THE RESCUE? (1)

```
public class Troll extends Basher {

    private Shouter shouter;

    public Troll(int bashRange, String phrase) {

        super(bashRange);
        shouter = new Shouter(phrase);

    }

    public String shout() {

        return shouter.shout();

    }
```

We implied before that HAS-A relationships might be of use here.

Can they **save** us in this case?

In many cases, we **prefer delegation to inheritance to avoid having to sacrifice our single extends for a small piece of functionality.**

Partially, yes, because by delegating actions inside `Troll` to either a `Shouter` or `Basher` object (while potentially inheriting from the other), we can exploit the **functionality** offered by **two** classes.

HAS-A RELATIONSHIPS TO THE RESCUE? (2)

```
public class Troll extends Basher {

    private Shouter shouter;

    public Troll(int bashRange, String phrase) {

        super(bashRange);
        shouter = new Shouter(phrase);

    }

    public String shout() {

        return shouter.shout();

    }
```

This also doesn't read particularly well.

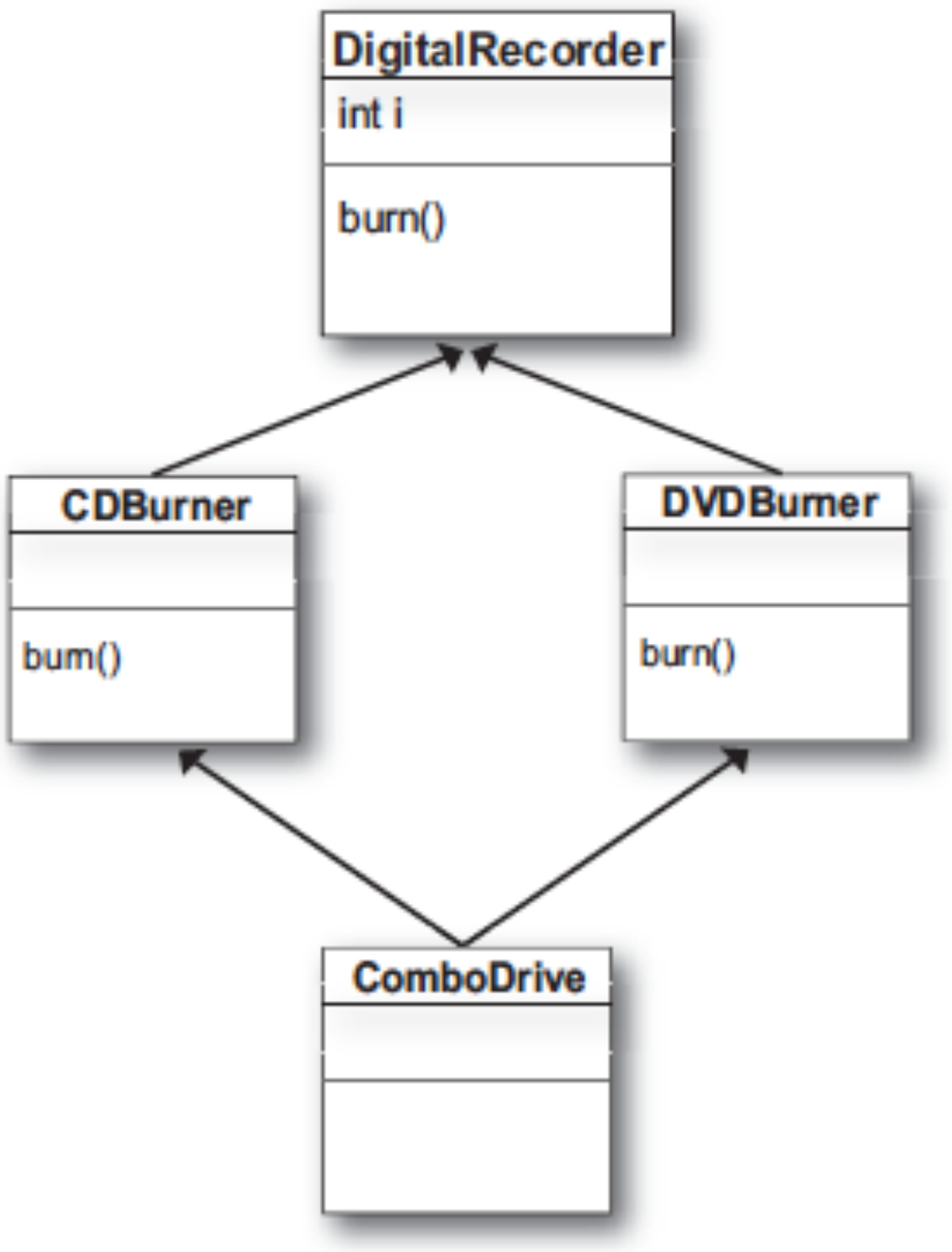
We implied before that HAS-A relationships might be of use here.

Can they **save** us in this case?

In many cases, we **prefer delegation to inheritance to avoid having to sacrifice our single extends for a small piece of functionality.**

But, in this case, the troll still cannot use the services offered by both `intimidate` and `destroy` because it **is** still only **one** of these things (**it bashes like a troll but doesn't shout like a troll**).

THE PROBLEM WITH MULTIPLE INHERITANCE: IN SHORT



Which version of burn does ComboDrive get?

What's holding us back from extending two classes (Option 2)?

Multiple inheritance can lead to **ambiguities**, if two methods exist in **different superclasses** with the same name and **different functionality** (more detailed explanation next term).

REMEMBER: CLASS INHERITANCE (2)

One class **inherits** (**copies**) **all of the code** from another, thus assuming the same **interface**.

Because a class's interface determines its identity, the inheriting class is now **also** considered to be of the **same type** as the class it inherits from.

- We therefore achieve our desired **IS-A** relationship.

EMPTY METHODS? (1)

```
public class Basher {

   public int getBashRange(){}

}
```

```
public class Shouter {

   public String shout(){}

}
```

What if all the classes we inherited from **didn't have any code in their methods**?

We wouldn't be able to **reuse** any functionality, but this would allow us to avoid the **inheritance ambiguity** listed on Slide 116, because we would **always add functionality in the child class** via **overriding**.

EMPTY METHODS? (2)

```
public class Basher {

   public int getBashRange(){}

}
```

```
public class Shouter {

   public String shout(){}

}
```

Instead, all that would be inherited would be the **signatures** of the methods, plus their **return type** and **access modifier** (i.e. the class **interface**) **but** this would be **fine** as this is enough for a class to share the **identity** of another class, and thus **be** another class.

EMPTY METHODS? (3)

```
public class Basher {

   public int getBashRange(){}

}
```

```
public class Shouter {

   public String shout(){}

}
```

In other words, this would **still** allow us to express (both formally and conceptually) that one class **IS-A**(nother) class:

```
public class Troll extends Basher {
```

Moreover, there would now be **no argument against multiple inheritance** (Option 2):

```
public class Troll extends Shouter, Basher {
```

Although this won't yet compile.

IMPLEMENTING AN INTERFACE (1)

```
public interface Basher {

   public int getBashRange();

}
```

```
public interface Shouter {

   public String shout();

}
```

Because, when a class is empty in this way, all that would be inherited from that class would be its **interface**, we signify this to Java **formally** by calling a class with empty methods an… **interface**.

- Note the ability to place semi-colons after methods, which we cannot do in a normal class.

In addition, classes **implement** interfaces, rather than extending them:

```
public class Troll implements Shouter, Basher {
```

ASIDE: INTERFACE VS. INTERFACE

In the previous slide we used the word interface in two different ways:

- To **conceptually** refer to the public facing functionality of a class; the **methods** it **exposes**; what a **user** of that class **interacts** with.
- To refer to a type of class that has the sole purpose of permitting another class to **be its type** in a **non-limiting** way (i.e. the implementing class can also implement other interfaces; **be other things** but **not function** like other things), by presenting **just the interface** of a class.

Hopefully the relationship between these two ideas is clear.

And hopefully it's clear that this has **nothing** to do with **graphical user interfaces**.

INTERFACES: RULES

```
public interface Basher {

    public int getBashRange();

}
```

```
public interface Shouter {

    public String shout();

}
```

Why do we need to **formally** indicate to Java that a class is being used as an **interface?**

Because there are then **rules** for the form of this class.

INTERFACES: RULE 1 - EMPTY METHODS

```
public interface Basher {

   public int getBashRange();

}
```

```
public interface Shouter {

   public String shout();

}
```

The most natural rule, if a class is an interface, is that its methods **must be empty.**

- If methods are not empty, the interface **won't compile**, to avoid functionality conflict.

Caveat: *This restriction was lifted in Java 8 in favour of marking methods in an interface with default functionality as* ***default*** *in order to identify potential function ambiguity in advance. I find this both conceptually awkward, and more likely to introduce compiler errors, so I would recommend omitting default functionality from your interfaces.*

INTERFACES: RULE 2 - NO OBJECTS

```
public interface Basher {

   public int getBashRange();

}
```

```
public interface Shouter {

   public String shout();

}
```

Because interfaces don't contain any functionality, it **doesn't** make sense for Java to permit **objects to be made** of these classes.

- Therefore interfaces can't include **constructors** or anything else that is associated with making **instances** of classes.

INTERFACES: RULES 3 AND 4 - PUBLIC AND FIELDS

```
public interface Basher {

   public int getBashRange();

}
```

```
public interface Shouter {

   public String shout();

}
```

Everything in the interface has to be **public**.

- **Accessible** (public) **methods** define the (conceptual) **interface** of a class, which is all that is provided to implementing classes, so it wouldn't make sense to permit private methods.

State cannot be represented in an interface, but can be added by an implementing class.

- (Public) fields are permitted, but values must be assigned on the field because they are **static by default** (as you cannot make instances of an interface) and thus **final** (to avoid global updates) by default. Used to define **constants** (contentious).

INTERFACES: RULE 5 - METHOD IMPLEMENTATION

```
public interface Basher {

   public int getBashRange();

}
```

```
public interface Shouter {

   public String shout();

}
```

In addition, when we mark a class as an interface and a class implements that interface, the methods themselves are **not actually inherited** (copied).

Instead what is inherited is an **obligation for the implementing class to add these methods**. If it does not, then the code will not compile.

- **Functionality** is still added to these methods in the implementing class, negating any issue with two interface methods **sharing the same signature**.

THE FUN OF A TROLL (2)

```
public class Troll implements Shouter, Basher {

    private String phrase;
    private int bashRange;

    public Troll(String phrase, int bashRange) {

        this.phrase = phrase;
        this.bashRange = bashRange;

    }

    public String shout() {

        return phrase;
    }

    public int getBashRange() {

        return bashRange;

    }
}
```

To implement a method from an interface you do so in the same way as ***overriding*** *a method from a superclass.*

If Troll did not re-implement shout and getBashRange, this class would not compile.

```
public interface Basher {

    public int getBashRange();

}
```

```
public interface Shouter {

    public String shout();

}
```

Here we are saying that a Troll **IS-A** Shouter and **IS-A** Basher so it has to **behave** like it.

- What are the benefits of this?

NON-EXCLUSIVE SERVICE PROVIDERS (1)

The `intimidate` and `destroy` methods now have the option to accept objects that **implement an interface**, **rather** than **extend a class**, and to thus be **non-exclusive** (Option 3).

- To use these services, a class only has to **implement** an interface, not **tie itself into an entire hierarchy**, and a class can implement as many interfaces as it likes.

At the same time, these methods know that only an object that implements the interface type (e.g. `Shouter` and `Basher`), can be passed as an argument.

```
public void intimidate(Shouter shouter) {

    System.out.println(shouter.shout());

}
```

Village.java

```
public void destroy(Basher basher) {

    houses -= basher.getBashRange();

}
```

Village.java

NON-EXCLUSIVE SERVICE PROVIDERS (2)

As we know, an object that implements an interface **must** implement all the methods from that interface.

Thus, like traditional **polymorphic arguments**, these methods **can call methods on the passed object from the interface type (e.g.** shout **and** getBashRange**) in order to achieve their advertised service, and be sure that those methods will exist**.

- Intimidate and shout would not be able to have this assurance if interfaces did not ensure the existence of their methods in implementing classes.
- But of course, appropriate functionality is still not guaranteed.

```
public void intimidate(Shouter shouter) {
    System.out.println(shouter.shout());
}
```

Village.java

```
public void destroy(Basher basher) {
    houses -= basher.getBashRange();
}
```

Village.java

THE FUN OF A TROLL (3)

```
public class Troll implements Shouter, Basher {

    private String phrase;
    private int bashRange;

    public Troll(String phrase, int bashRange) {

        this.phrase = phrase;
        this.bashRange = bashRange;

    }

    public String shout() {

        return phrase;
    }

    public int getBashRange() {

        return bashRange;

    }
}
```

And now the troll is free to intimidate and bash as he pleases, along with his friends, should they wish to:

```
public void intimidate(Shouter shouter) {

    System.out.println(shouter.shout());

}
```

Village.java

```
public void destroy(Basher basher) {

    houses -= basher.getBashRange();

}
```

Village.java

```
public class Werewolf implements Shouter {
```

```
public class Giant implements Basher {
```

As a final optimisation, `Troll` (and `Werewolf` and `Giant`) could still extend `Enemy`, and thus fight with villagers, while also shouting and bashing:

```
public class Troll extends Enemy implements Shouter, Basher {
```

If `Enemy` *were to have a method (Method A) with the same signature as a method in either the* `Shouter` *or* `Basher` *interfaces (Method B), Method A would be considered the implementation of Method B, and thus issues with functionality ambiguity are still avoided.*

IMPLEMENTING AN INTERFACE VS. EXTENDING A BASE CLASS (1)

Are interfaces now preferable to normal classes?

- For example, should our methods now **only** accept interfaces for maximum **flexibility**?

Base classes (that can be extended) and interfaces play **different roles**:

- Interfaces present **abstract behaviours**; **capabilities**; things that can be **done** (e.g. shouting and bashing), **not** the **functionality** supporting that behaviour; base classes **define a single existence,** and thus both **behaviour** and **functionality** which can be **copied**.
    - This tradeoff allows for classes to **be** multiple things (i.e. implement multiple interfaces), and thus use **multiple services**, at the **cost** of **not being able to reuse functionality**.

A MORE PRACTICAL EXAMPLE?

Trolls, villagers and Martin playing on his Playstation too much is all very well, but what about a more **practical example** of the use of **interfaces and services**?

Some library classes that provide services don’t want to **force** one of our classes to inherit a **single other class** in order to use their service. Therefore, they instead accept objects of an **interface type**.

```
private TreeMap<String, Integer> customerTotalSpend;
```

A `TreeMap` provides us with a **service**: it will **sort** any `<key,value>` pairs we add **by key**.

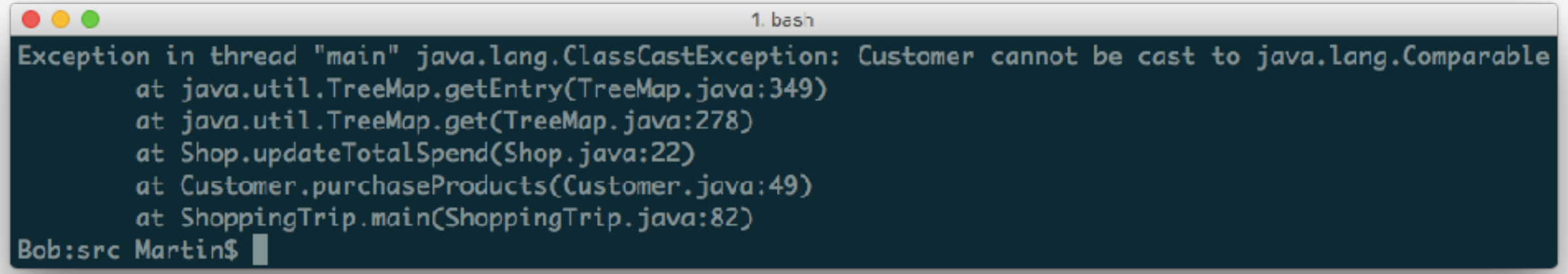
But to do so, we have to offer that service a **method** (a behaviour) that enables it to sort our keys.

`TreeMap` **quantifies this request** by forcing any supplied keys to be comparable, but retains the **flexibility** for keys to **extend other classes** by accepting keys that **implement the Comparable interface**.

We have seen what happens if we do not adhere to this restriction:

ERROR: CLASS CAST EXCEPTION AND COMPARABLE

```
private TreeMap<Customer, Integer> customerTotalSpend;
```



Strings implement the Comparable interface by default but Customer (one of our own classes) does not, hence this error.

Our Customer class will not be able to correctly support the service offered by the TreeMap class (i.e. customers cannot be compared), and the TreeMap class knows this.

COMPARABLE

Comparable is an interface that **ensures** implementing classes **supply** a single method that returns an integer to indicate whether a compared object should be **positioned** before or after the current one when being **sorted**.

java.lang

Interface Comparable<T>

Type Parameters:

`T` - the type of objects that this object may be compared to

Method Summary

Methods

Modifier and Type	Method and Description
`int`	**`compareTo(T o)`** Compares this object with the specified object for order.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Other classes, such as `TreeMap`*, call* `compareTo` *in order to achieve their service, in the same way that* `contains` *calls* `equals`.

*This is how something should **behave** if it wants to **be** comparable.*

IMPLEMENTING COMPARABLE (1)

```
public class Customer implements Comparable<Customer> {
```

Like a `Troll` that must `shout` if it wants to be a `Shouter` (and `intimidate` a village) and provide a `bashRange` if it wants to be a `Basher` (and `destroy` a village), a `Customer` must provide a `compareTo` method if is to be `Comparable` (and sorted in a `TreeMap` (or other places)):

```
public int compareTo(Customer o) {



}
```

Customer.java

ASIDE: GENERIC INTERFACES

```
public class Customer implements Comparable<Customer> {
```

You will have noticed that we were able to *type* the `Comparable` interface in the previous example in the same way that we can *type* lists, using the diamond notation.

Doing this effectively **alters** the **form** of the `compareTo` method in the `Comparable` interface to include a parameter of our own `Customer` type, thus we can implement (override) this method without having to check the instance of the parameter, or cast it:

```
public int compareTo(Customer o) {
```

We didn't have the same flexibility when overriding `equals` through inheritance, and it would be cumbersome to *type* object anyway.

IMPLEMENTING COMPARABLE (2)

In the case of our customer, we don't need to do too much work in our `compareTo` method, as there's only really one field upon which we can base our comparisons: `name`, which is a string.

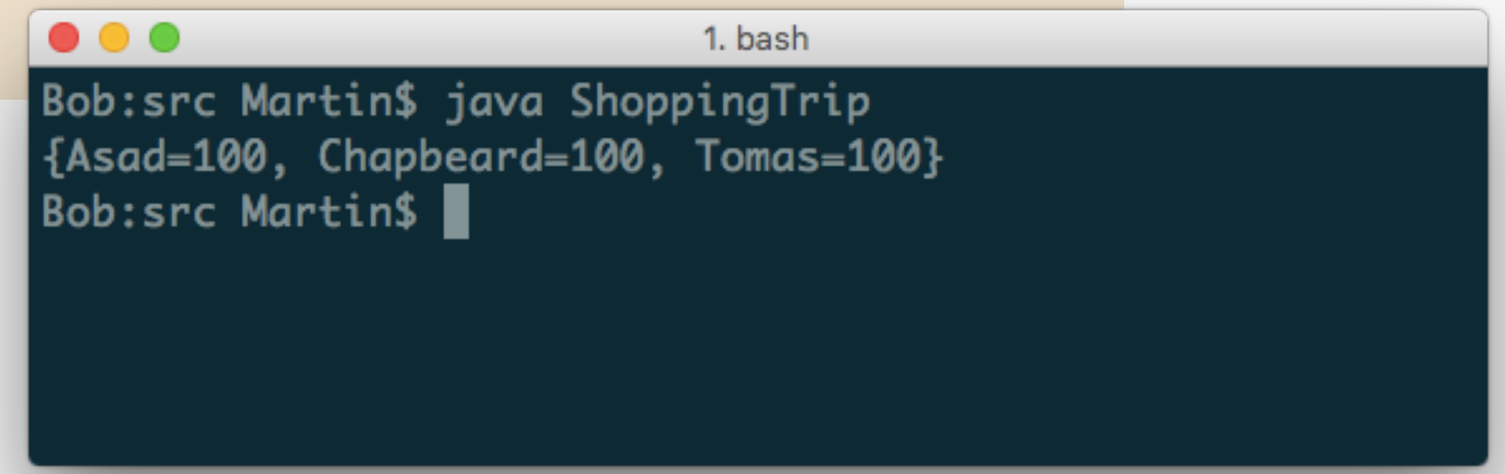
Therefore, we can pass on the task of comparison to the `String` class:

```
public int compareTo(Customer o) {

   return name.compareTo(o.name);

}
```
Customer.java

Customers are now `Comparable`, so `TreeMap` knows our `Customer` class can support the service it offers.

THE SERVICE PROVIDED BY TREEMAP

```
shop.addProduct(diamond);
shop.addProduct(crownJewels);
shop.addProduct(silverLocket);

Customer martin = new Customer("Chapbeard");
Customer tomas = new Customer("Tomas");
Customer asad = new Customer("Asad");

shop.updateTotalSpend(martin, 100);
shop.updateTotalSpend(tomas, 100);
shop.updateTotalSpend(asad, 100);

System.out.println(shop.getCustomerTotalSpend());
```



IMPLEMENTING AN INTERFACE VS. EXTENDING A BASE CLASS (2)

Some more differences between implementing an interface compared to extending a base class:

- It can be difficult to **modify** an interface as all implementing classes have to change too; it can be easier to modify a base class, as the implementation restrictions aren't explicit.
- The behaviours expressed in an interface pertain to those behaviours that are likely to **span multiple inheritance hierarchies** (consider a `Villager` and a `Enemy`, both of which are likely to `shout`); the behaviour expressed in a class are only shared by those classes **within that hierarchy**.

IMPLEMENTING AN INTERFACE VS. EXTENDING A BASE CLASS (3)

Some more differences between implementing an interface compared to extending a base class:

- To use of an interface type can support the **flexibility of a program** by allowing the programmer to **change the underlying implementation** of entities such as data structures; base classes do not offer this support as **explicitly** (because you can make and store an object of the base class).

- Interface behaviours **vary** with **each implementation** (e.g. things all `shout` in different ways); behaviours in a base class are likely to be **shared between implementations** (e.g. having and setting a `name`).

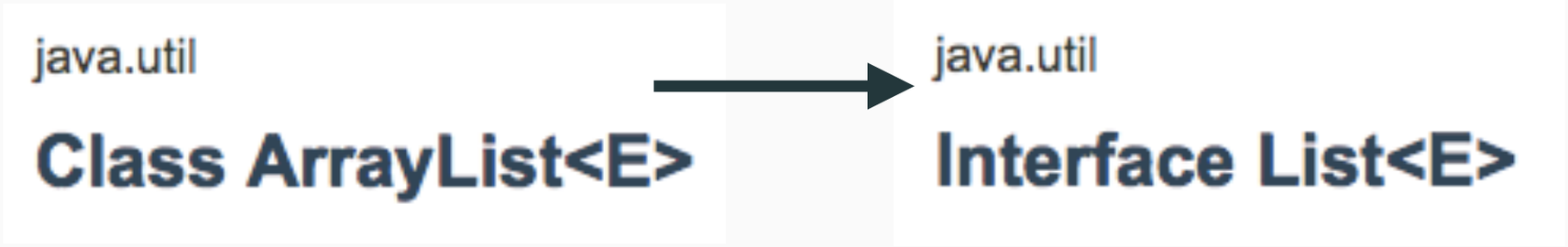
REMEMBER: TOPIC 8 MYSTERIES

Let's look at these last two points in more detail by considering some things we saw in Topic 8:

We have left a few mysteries for later:

- Why is a TreeSet a Set, or a TreeMap a Map? What does this mean?
- Why did we not use or instantiate a List or Map? Turns out that we can't, but why?

Lots of the collections we have seen have their **behaviour expressed** via an interface:



This is why we cannot make objects of the List type.

Why? Based on what we've discussed:

- For **services**: a method in a class that wants to offer a service using a collection (e.g. `List`), can accept any type of this collection (e.g. `ArrayList`) as an argument, without having to force single inheritance.
- But also for the reasons seen on Slide 143:

SWAPPING THE UNDERLYING IMPLEMENTATION (1)

```
public class Customer implements Comparable<Customer> {

   private ArrayList<Product> shoppingBasket;
   …

   public Customer(String name) {

      shoppingBasket = new ArrayList<Product>();
      …

   }

   public ArrayList<Product> getShoppingBasket() {

      return shoppingBasket;

   }
```

What if I want to store things in a different format?

SWAPPING THE UNDERLYING IMPLEMENTATION (2)

```
public class Customer implements Comparable<Customer> {

   private LinkedList<Product> shoppingBasket;
   …

   public Customer(String name) {

      shoppingBasket = new LinkedList<Product>();
      …

   }

   public LinkedList<Product> getShoppingBasket() {

      return shoppingBasket;

   }
```

Lots of changes!

SWAPPING THE UNDERLYING IMPLEMENTATION (3)

```
public class Customer implements Comparable<Customer> {

   private List<Product> shoppingBasket;
   …

   public Customer(String name) {

      shoppingBasket = new ArrayList<Product>();
      …
   }

public List<Product> getShoppingBasket() {

   return shoppingBasket;

}
```

*This is therefore another example of **polymorphism**, related to Java's data structures.*

SWAPPING THE UNDERLYING IMPLEMENTATION (4)

```
public class Customer implements Comparable<Customer> {

   private List<Product> shoppingBasket;
   …

   public Customer(String name) {

      shoppingBasket = new LinkedList<Product>();
      …
   }

public List<Product> getShoppingBasket() {

   return shoppingBasket;

}
```

*If we want to change how this interface is **realised** we can do so trivially.*

We call this **coding to the interface**, and it is a **style choice**.

UNIQUE BEHAVIOUR IN EACH IMPLEMENTATION

In our village, `Basher` and `Shouter` feature behaviours that are **always realised in different ways**. That is, different entities (such as trolls, werewolves and giants), always `shout` or `(get)Bash(Range)`, differently.

- This lack of **default functionality**, or **unique functionality in each implementation**, justifies the use of an interface.

Similarly, a List or a Set feature behaviours that are **usually** realised in different ways. That is, different collections (such as `ArrayList` and `LinkedList`), usually present concrete functionality (e.g. `get` and `size`) in different ways.

- **Caveat:** *Usually* is the key word here. In reality, things are a little more complex...

A quick detour into Abstract Classes...

PARTIAL FUNCTIONALITY? (1)

In reality, for **certain behaviours** (methods) in the `List` interface, there is functionality that **could be shared** between the classes that implement this interface (e.g. the behaviour of `subList`).

- For some behaviours, all implementing classes **can** function in the same way.

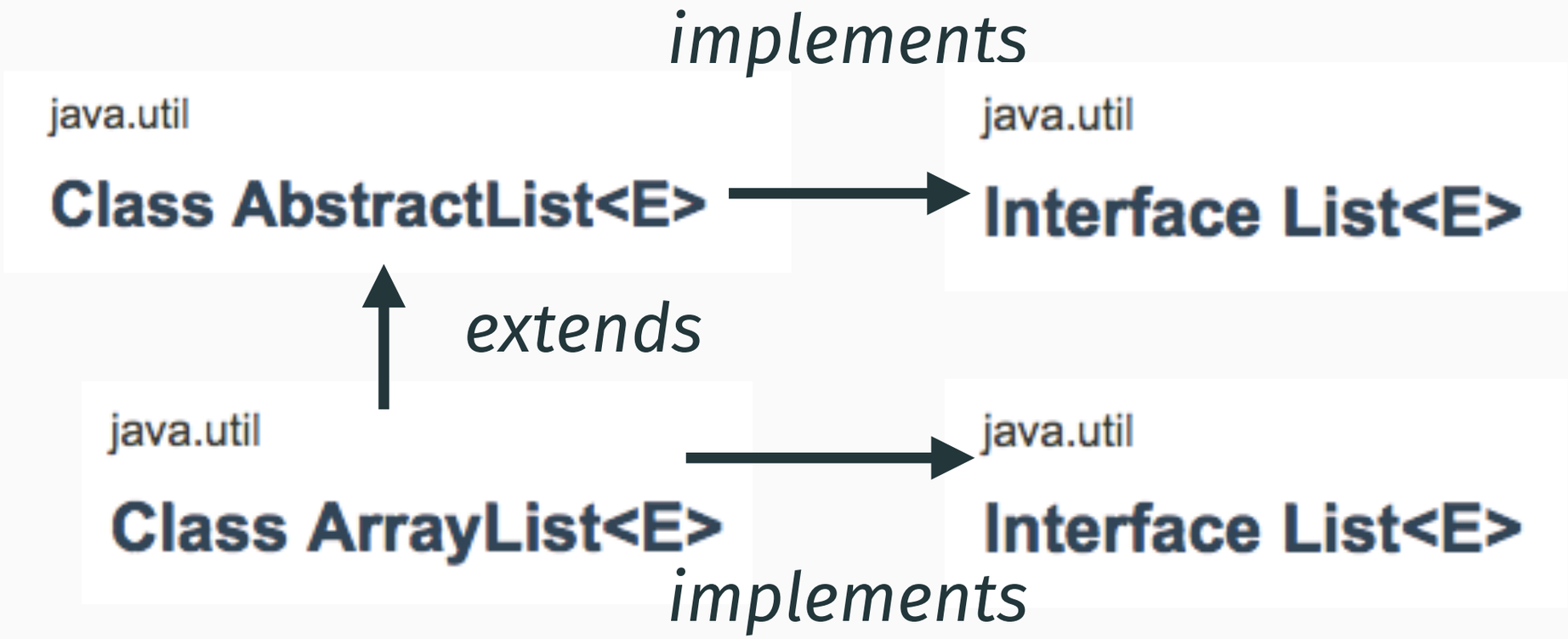
This might suggest that `List` should be a **normal base class**, such that this functionality can be **inherited**.

However, the motivation for keeping `List` (and other **collection types**) as interfaces remains: offering non-exclusive services, the ability to switch implementations, and indeed a majority of the behaviours are still unique for each implementation.

What's the solution? What we really need is **partial functionality**.

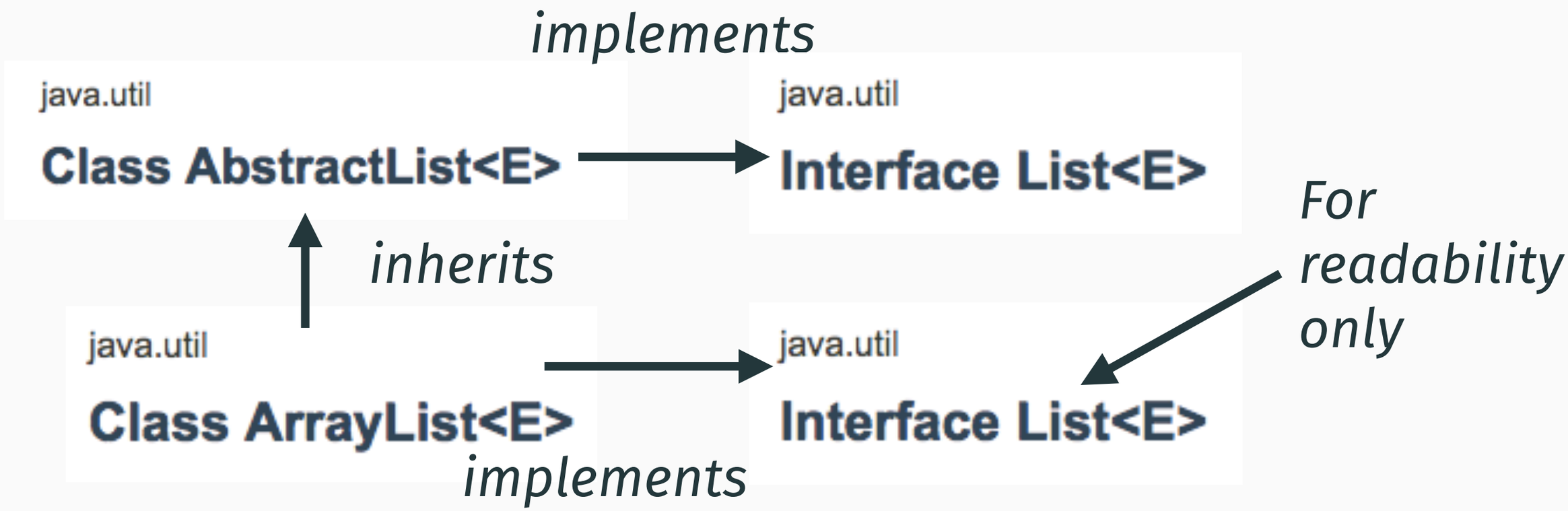
PARTIAL FUNCTIONALITY? (2)

The solution, in order to introduce partial functionality, is as follows:



Here, `AbstractList` sits **in-between** the `List` interface and the `ArrayList` implementation, **filling in some functionality**, while **passing on the constraints to complete other functionality**.

PARTIAL FUNCTIONALITY? (3)



In this way, `ArrayList` is still of `List` type (because `AbstractList` implements `List`), and takes the **benefits** of being as such (e.g. service use), but it still able to inherit **default functionality** from `AbstractList`, or reimplement it as it sees fit.

In reality the `ArrayList` *class is a bit of brat and decides to reimplement almost everything from* `AbstractList`*, but it, and other lists, still have the* ***option*** *to* ***reuse functionality****.*

ABSTRACT CLASSES

The reasons `AbstractList` is able to function in this way is because it is something called an **abstract class**.

An abstract class can be seen as half way between an interface, and a base class: **some methods are abstract** (empty, like in an interface); some are not.

- Methods marked abstract are designed to be **reimplemented**;
  normal methods are designed to **share functionality**.

```
public abstract class AbstractList implements List {

    abstract public get(int index);

    public List subList(int fromIndex, int toIndex) {
        return (this instanceof RandomAccess ?
        . . .
    }
```

ABSTRACT CLASSES: CLASS FEATURES

Like normal base classes, abstract classes:

- Can have **non-abstract** methods with functionality.
- **Constructors** (that must be invoked with a super call if necessary).
- **Non-static** and **non-final** fields.

Because of these things, like a base class, **only one abstract class can be extended**.

ABSTRACT CLASSES: INTERFACE FEATURES

Like interfaces, abstract classes:

- Have methods without functionality, however these methods are marked **abstract** in order to differentiate them from the non-abstract methods that exist in the same class.
- When you extend an abstract class, **an implementation must be provided** for all of the abstract methods from the class (unless the extending class is also abstract).

Due to their (potential) partial implementation, abstract classes cannot be instantiated.

WHY ARE ABSTRACT CLASSES USEFUL? (1)

Therefore, abstract classes **aren't** designed to solve the multiple inheritance problem, in the same way as interfaces.

Instead, they are:

- Designed to **pass on common functionality** in the presence of some **unknown** functionality (`subList` (common) vs `get` (unknown)).

- Designed to capture when there **shouldn't be objects of a certain class** (more in a moment).

WHY ARE ABSTRACT CLASSES USEFUL? (2)

Therefore, abstract classes **aren't** designed to solve the multiple inheritance problem, in the same way as interfaces.

Instead, they are:

- Designed to sit in-between an interface and an implementing class, in order to create **partial functionality** (e.g. the relationship between `ArrayList`, `AbstractList` and `List`).

- It is because objects cannot be made of an abstract class that it is able to **safely ignore** the restrictions of an interface, which is that all methods must be implemented.

- Instead, this restriction is passed on to any **subclasses** of the abstract class (e.g. `ArrayList` must provide an implementation for get). Again, unless that class is also abstract..

- Partial functionality can still be **replaced**, if needed via **overriding** (which `ArrayList` chooses to do).

ASIDE: WHY WOULD I WANT TO INHIBIT INSTANCES OF A CLASS? (1)

When modelling the game of Battleship, we might decide to organise the setting of a ship's size by providing a superclass that sets up parts etc., and this functionality can then be called from the subclasses (including Battleship).

```
public class Ship {

   private ArrayList<Part> parts;

   public Ship(int row, int size) {

      parts = new ArrayList<Part>();

      for ( int i = 0; i < size; i++ ) {

         parts.add(new Part(row, i));

      }

   }
```

ASIDE: WHY WOULD I WANT TO INHIBIT INSTANCES OF A CLASS? (2)

But what is a **ship**? It doesn't have a **size**, so it's not really a **thing**; it's just **part of an idea.**

- We also wanted **encapsulate** the number of parts as ship has.

This can be captured by making a `Ship` class **abstract** (with optional abstract methods, if we wanted) such that it is only designed to be **subclassed**, and thus realised (i.e. when it is given parts):

```
public abstract class Ship {
```

Credit to you guys for leading me towards this example.

```
public class Battleship extends Ship {

    private ArrayList<Part> parts;

    public Battleship(int row) {

        super(row, 5);

    }
```

Back to interfaces...

WHY WOULD WE WANT TO USE INTERFACE IN OUR OWN CODE?

We've talked a lot about using the interfaces designed by other people.

But why we would want to build our own interfaces?

Lots of the reasons are the same:

- To encourage classes to **support behaviours** that allow them to use our services.
- Conceptually, we may want to define an entity that always has **unique behaviour in each implementation**.
- For implementation flexibility.

ASIDE: CREATING OUR OWN INTERFACES: HIDE-AND-SEEK

```
public class HiddenObjectGraph<V, E extends DefaultWeightedEdge> extends SimpleWeightedGraph<V, E> {

    public boolean fromVertexToVertex(GraphTraverser traverser, V sourceVertex, V targetVertex) {

        if ( containsEdge(sourceVertex, targetVertex) ) {

            E traversedEdge = getEdge(sourceVertex, targetVertex);

            double actualCost = this.getEdgeWeight(traversedEdge);

            // The unique cost to the traverser, based upon their traversals so far
            double uniqueCost = 0.0;
```

I wanted people to be able to use the services offered by my method method (exploring a graph), so I offered an interface for implementation:

```
public interface GraphTraverser extends Comparable<GraphTraverser>, Runnable {

   public HashSet<StringVertex> uniquelyVisitedNodes();

   public HashSet<StringEdge> uniquelyVisitedEdges();

   public ArrayList<StringVertex> allHideLocations();
```

INTERFACES: GOING FORWARD

Next semester you will use interfaces when dealing with **events** in **graphical user** interfaces (just to make things confusing).

Here's a sneak preview:

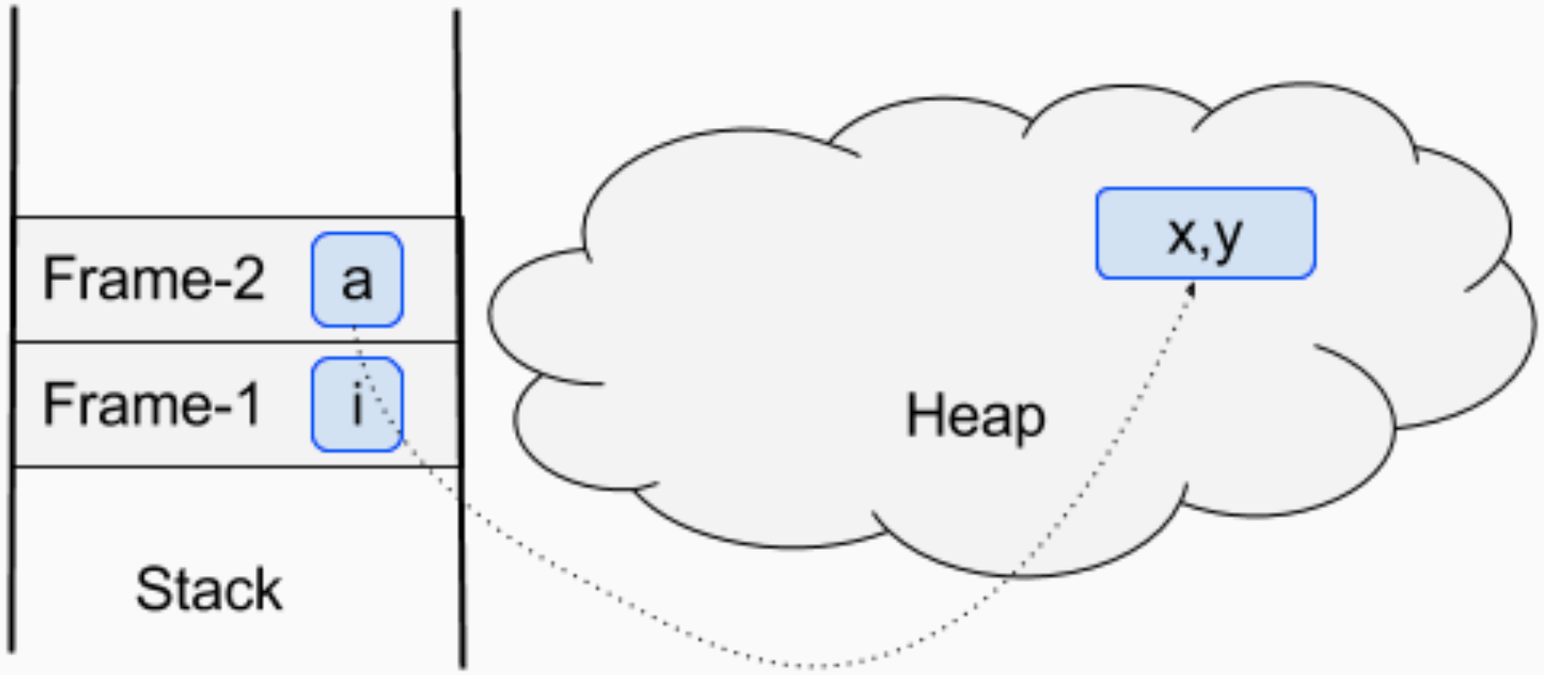
A service that calls a method in an interface type when the button is pressed.

```
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Boo!");
        }
});
```

An implementation of that interface.

REMEMBER: ASIDE: OBJECTS IN MEMORY

We will use the definition of an object quite loosely in the first semester of PPA, in order to **simplify** things, and help your initial understanding.



We'll look at this in more detail next semester.

In reality, unlike primitive values, objects are stored in another area of the JVM's memory known as the **heap**. The variable containing the copy of the class is actually a **memory reference** from a variable on the stack to an object on the heap.

Topic 9: Combining Objects

Programming Practice and Applications (4CCS1PPA)

Dr. Martin Chapman
Thursday 1st December 2016

programming@kcl.ac.uk
martinchapman.co.uk/teaching

These slides will be available on KEATS, but will be subject to ongoing amendments. Therefore, please always download a new version of these slides when approaching an assessed piece of work, or when preparing for a written assessment.