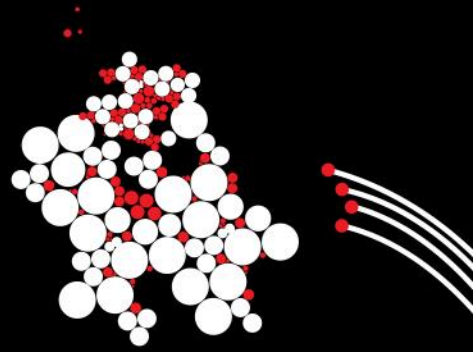
UNIVERSITY OF TWENTE.

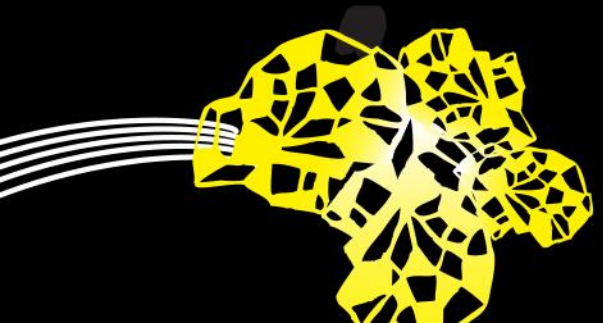


P3.1: INTERFACES AND INHERITANCE
P3.2: SUBTYPING

TOM VAN DIJK

201700117-1B MODULE 2: SOFTWARE SYSTEMS

26 NOVEMBER 2019



PROGRAMMING LINE OVERVIEW

Week 1 Values and variables Control flow	**Week 2** Classes and objects Testing	**Week 3** Interfaces and Inheritance Subtyping Security 1
Week 4 Arrays and Lists List implementations Collections	**Week 5** Stream I/O and MVC Exceptions Security 2	**Week 6** Concurrency Project kick-off IDE Tips & Tricks
Week 7 Basic Networking Networking and Multithreading GUIs	**Week 8/9** Advanced Java facilities Test	**Week 10** Project Test resit

LAST WEEK

Last week in Programming

- Objects as instances of classes
- Encapsulation
  - public methods and constructors for “external behaviour”
  - private variables and methods for “internal behaviour”
- Testing

THIS WEEK

This week in Programming

- Subclasses (extending classes) and inheritance
- Abstract classes and interfaces
- Polymorphism
- Subtyping
- Inheritance vs composition

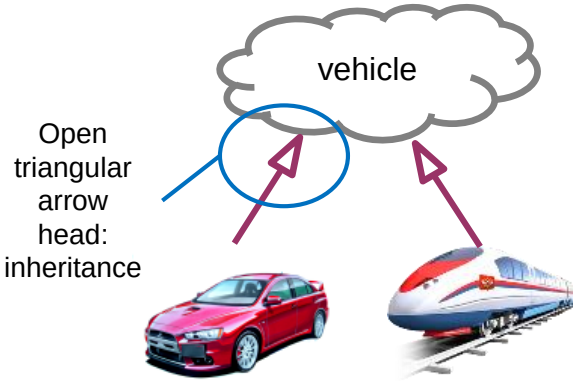
(Eck Sections 5.5 – 5.8)

PROGRAM DESIGN

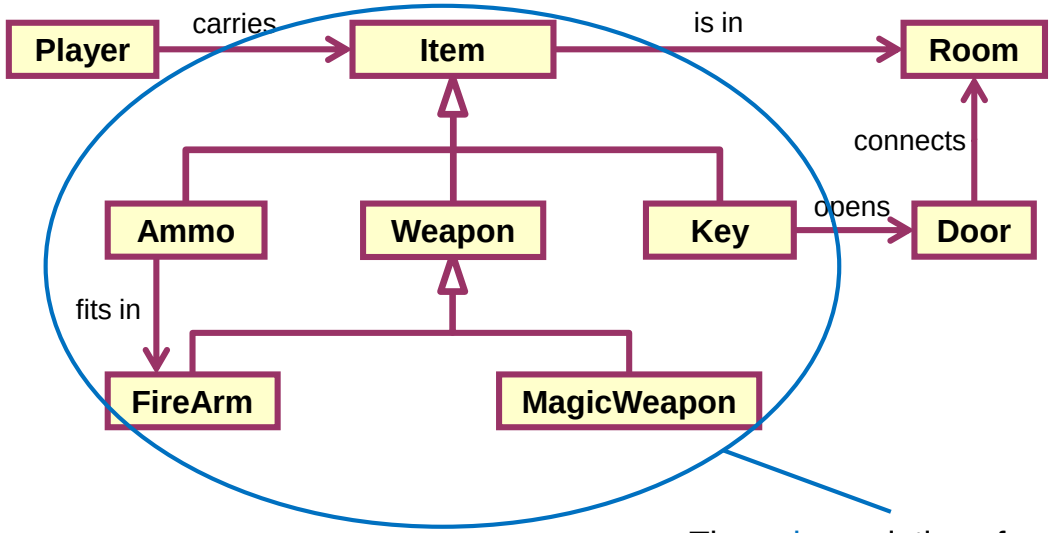
- Design software first, then concurrently test and implement
- Program design defines relations between concepts (classes)
- Last week: association (“has-a”, “belongs-to”, “occupies”)

ABSTRACTION

- Sometimes a B-object (instance of class B) can also be an A-object (instance of class A)
  - A car and a train also are a vehicle
  - Relation: B-object “is-an” A-object
- Synonyms
  - A is an abstraction of B
  - A generalises B
  - B specialises A
  - B implements A
  - B inherits from A
  - B extends A

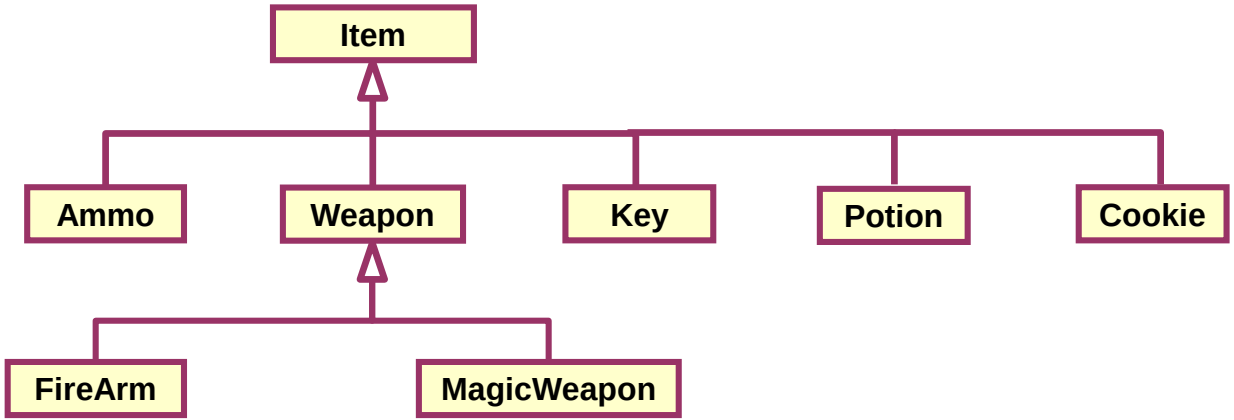


EXAMPLE: CLASS HIERARCHY IN A MAZE GAME



These is-a relations form a *class hierarchy*

EXAMPLE: WHY USE CLASS HIERARCHIES?



If the game code uses variables of type **Item**, say, for keeping an Inventory, then it is easy to add more types of “Items” later on, **without changing the code** for the Inventory**.**

EXAMPLE: JAVAFX LIBRARY (2D/3D GRAPHICAL USER INTERFACES)

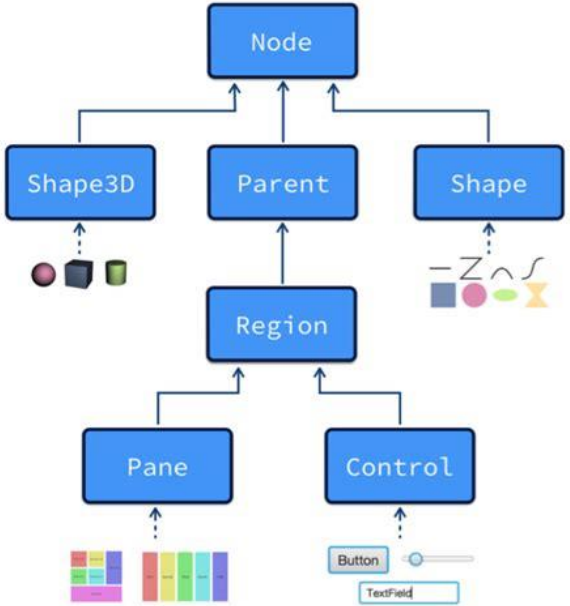


Image from DZone

Libraries like JavaFX contain many classes.

Typically, a lot of the “base code” is shared: put in common base classes like Node, or Shape.

For example

- A Node has a location, rotation, scale
- A Shape is filled or not filled

EXTENDING CLASSES

- A subclass extends a superclass

```
public class Dog extends Animal {

}
```

- Subclasses inherit (reuse) the class members of the parent class
  - methods of the child class can access public/protected members of the parent class
  - methods of the child class cannot access private members of the parent class
  - variables of the child class hide variables with the same name of the parent class
  - methods of the child class override methods with the same signature of the parent class
  - use this for the current object and super for members of the parent class
  - use super(…) in the constructor to invoke the constructor of the parent class

INHERITANCE

```java
public class A {
    private int field;

    public void doA() {
        field = field+1;
    }

    public int getField() {
        return field;
    }
}
```

```java
public class B extends A {
    private boolean field2;

    @Override
    public void doA() {
        field2 = getField() < 0;
        super.doA();
    }

    public boolean doMore() {
        return field2;
    }
}
```

The `@Override` annotation is “optional”, but you should always use it.
It warns you when making an error in the typing.

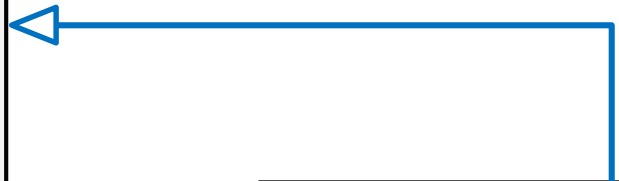
INHERITANCE EXAMPLE

```
public class Item {
    private Room place;

    public Item(Room place) {
        this.place = place;
    }

    public Room getPlace() {
        return this.place;
    }

    public boolean isPortable() {
        return false;
    }
}
```

```
public class Key extends Item {
    private Door door;

    public Key(Room place, Door door) {
        super(place);
        this.door = door;
    }

    @Override
    public boolean isPortable() {
        return true;
    }

    public boolean opens(Door door) {
        return this.door.equals(door);
    }
}
```

INHERITANCE EXAMPLE

```java
public class Point2D {
   private int x; private int y;
   public void move(int x, int y) {
      this.x = x; this.y = y;
   }
   public void reset() { move(0, 0); }
   public int getX() { return this.x; }
   public int getY() { return this.y; }
}
```



```java
public class Point3D extends Point2D {
   private int z;
   public void move(int x, int y, int z) {
      move(x, y); this.z = z;
   }
   public void reset() {
      super.reset();
      this.z = 0;
   }
   public int getZ() { return this.z; }
}
```

OVERLOADING VS OVERRIDING

- Overloading (also called static polymorphism)
  - Methods in a class with the same name but different signature
    (actually: just different number or type of parameters)
  - Example: `move` in previous example
  - Use sparingly, and only if confusion is unlikely
- Overriding (also called dynamic polymorphism)
  - Methods of a subclass with the same signature of a method of the parent class
  - Use `@Override` tags in front of overriding method (not when overloading)
    - Improves maintainability: fewer mistakes! Good practice
- Notice: Python has much more dynamic polymorphism than Java because of weak typing

CONTRACTS FOR OVERRIDING METHODS

- Contract in supertype: general, weak enough to allow overriding

```
public interface ClosedFigure {
   /*@ ensures \result > 0; */
   public int circumference();
}
```

- Specialised contract in subtype: specific, concrete & stronger
  - The same or weakened precondition
  - The same or strengthened postcondition

```
public class Circle implements ClosedFigure {
   /*@ ensures \result == 2 * Math.PI * radius(); */
   public int circumference() { ... }
}
```

- Contract of original method is respected
  - Calling `circumference` on a `ClosedFigure` will meet expectations

CONSTRUCTORS REVISITED

- Method without a return type, with the same name as the class
- Constructors can be overloaded but are not inherited
- All classes have a constructor
  - When omitted, the class has an implicit (default) empty-argument constructor
- Every constructor calls another constructor
  - First line: `super(args);` or `this(args);` (no `new` or constructor name!)
  - When omitted, implicitly calls `super()` (most frequent case)
  - Only valid if superclass has a constructor without parameters!
- Good practices
  - Constructor initialises all fields that need a non-default value
  - Constructor does not perform extensive computation

INHERITANCE EXAMPLE

```java
public class Point2D {
   // ... (as before)
   protected Point2D() {
      // empty
   }
   public Point2D(int x, int y) {
      this();
      move(x, y);
   }
}
```

Empty constructor:
assigns default value (`0`) to all fields
`protected`: only meant for subclasses

Overloaded constructor:
calls **`this`** to invoke default behaviour

```java
public class Point3D extends Point2D {
   // ... (as before)
   public Point3D() {
      // empty
   }
   public Point3D(int x, int y, int z) {
      super(x, y);
      this.z = z;
   }
}
```

Empty constructor:
implicitly calls **`super()`** (visible!)

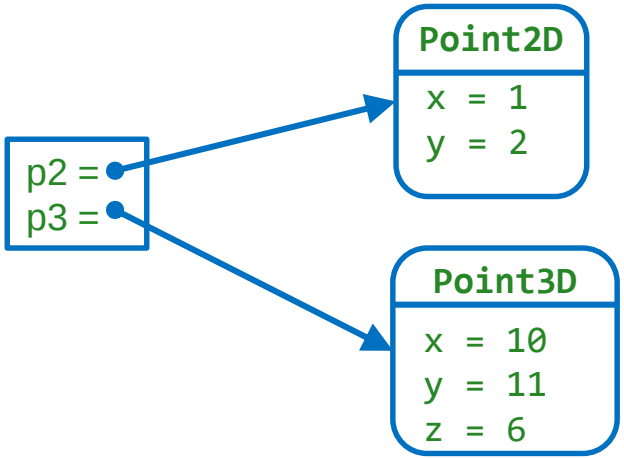
Overloaded constructor:
explicitly calls a **`super`** constructor

VISUALISING OBJECTS

- After the code snippet

```
Point2D p2 = new Point2D();
p2.move(1, 2);
Point3D p3 = new Point3D();
p3.move(4, 5, 6);
p3.move(10, 11);
```

  - what is in memory?



SUMMARY SO FAR

- Subclass and superclass and inheritance
- A class can extend at most one class
- Overriding, overloading, polymorphism

ABSTRACT METHODS AND CLASSES

- An abstract method is a method without a body

```
public abstract class SomeAbstractClass {
    public abstract void doSomething(int someNumber);
    protected abstract double computeSomething(int numberOne, double numberTwo);
}
```

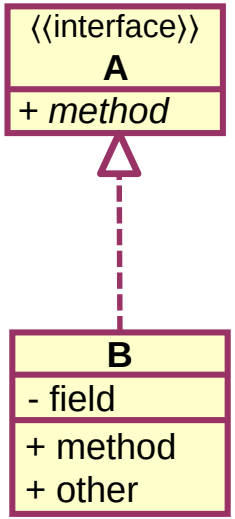
- Abstract methods must be in an abstract class
- Abstract classes are “incomplete”/“partial” and typically contain “base” functionality
- Subclasses must either implement the abstract methods or also be abstract
- You cannot instantiate an abstract class but an abstract class has a constructor

INTERFACES

- An interface is a special type of class: only specification, no implementation

- All variables are public final static
- All methods are public abstract

- Classes can extend one class and implement multiple interfaces
- Interfaces can extend multiple interfaces



INTERFACES

Very simple syntax

```
interface MyInterface {
    /**
     * Specification 1 ...
     */
    void myMethod1();   // <-- a semicolon instead of the method body

    /**
     * Specification 2 ...
     */
    int myMethod2(int i, int j);
}
```

INTERFACE IMPLEMENTATION: EXAMPLE

```
interface Item {
    Room getPlace();
    boolean isPortable();
}
```

```
public class Key implements Item {
   private Room place;
   private Door door;

   public Key(Door door) {
      this.door = door;
   }

   public Room getPlace() {
      return place;
   }
   public boolean isPortable() {
      return true;
   }

   public boolean opens(Door door) {
      return this.door.equals(door);
   }
}
```

implemented methods have same signature (method names, result, parameter types) and visibility

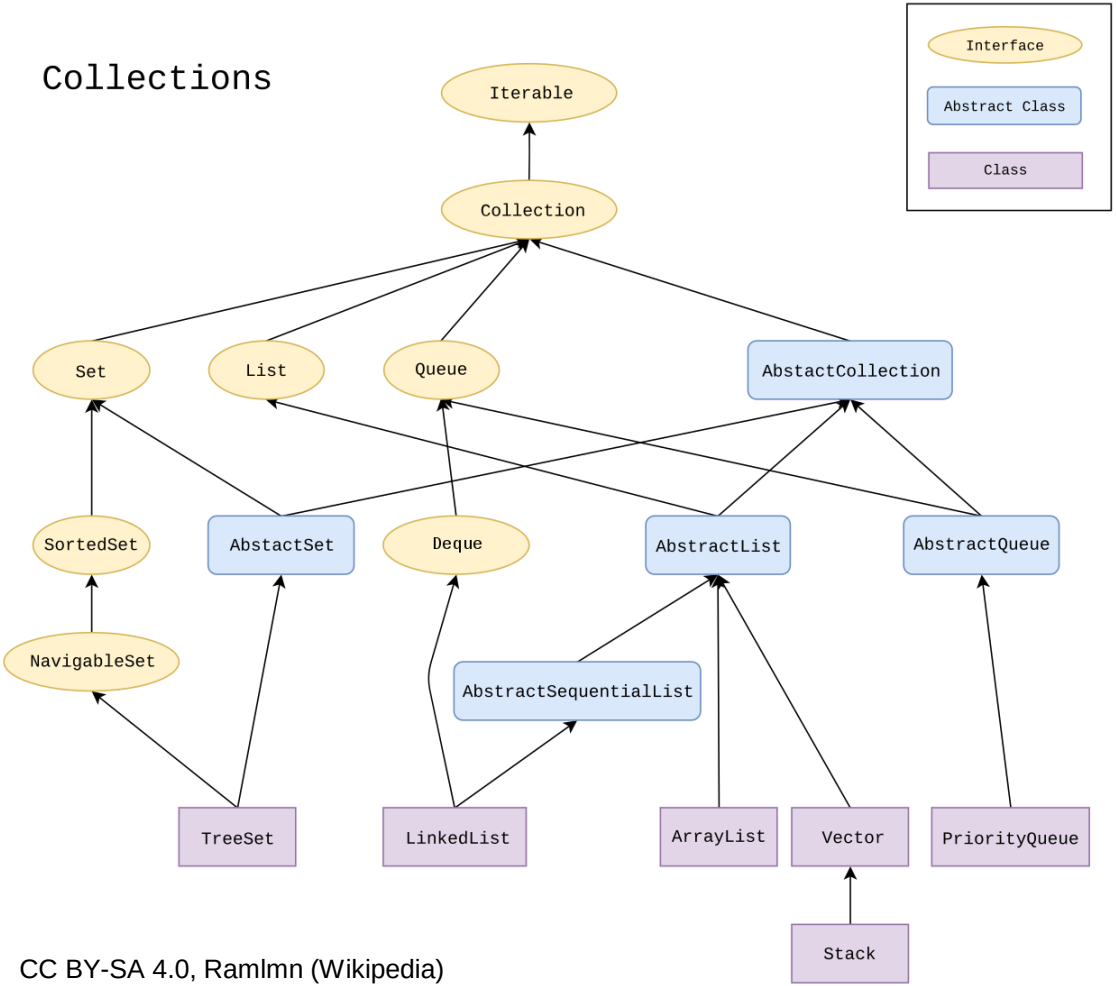
implementation has additional methods, fields and constructor

INTERFACES

- Java has no native list datastructures, only arrays
- The “Collections” library includes many useful datastructures including Lists
- All list types have a common interface List
- List defines many methods. Some of the more important ones are:

```
interface List {
    boolean add(Object e);
    boolean remove(Object e);
    boolean contains(Object e);
    Object get(int index);
    Object set(int index, Object e);
    Object remove(int index);
    int size();
}
```

Collections



INTERFACES

- A well known Java Interface is MouseListener

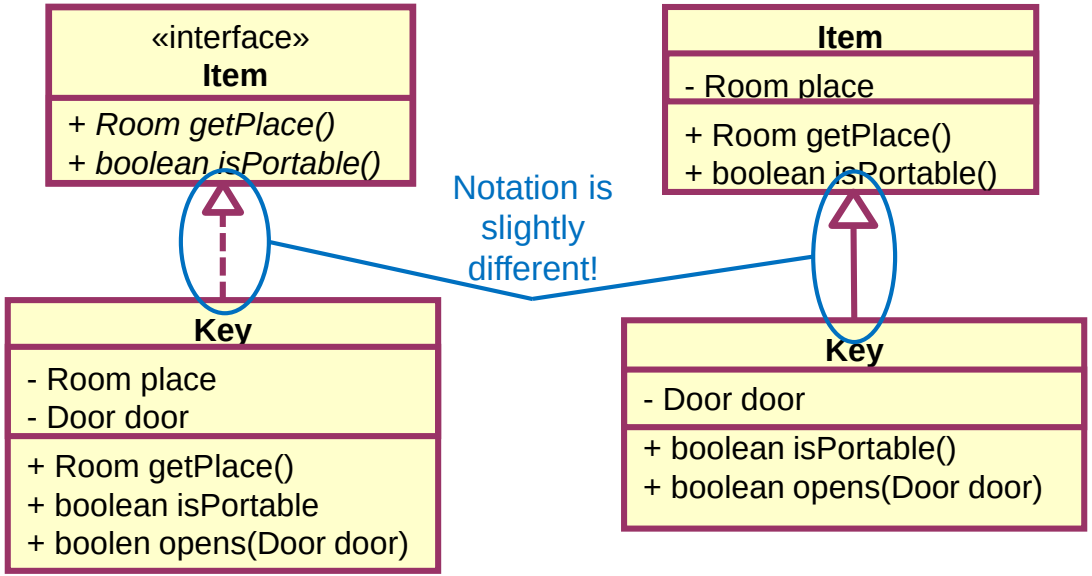
```
interface MouseListener {
    void mouseClicked (MouseEvent e);
    void mouseEntered (MouseEvent e);
    void mouseExited (MouseEvent e);
    void mousePressed (MouseEvent e);
    void mouseReleased (MouseEvent e);
}

public class MyClass implements MouseListener {
    // implement all five methods
}

theBigRedButton.addMouseListener(new MyClass());
```

EXAMPLE INTERFACE: java.util.Comparable

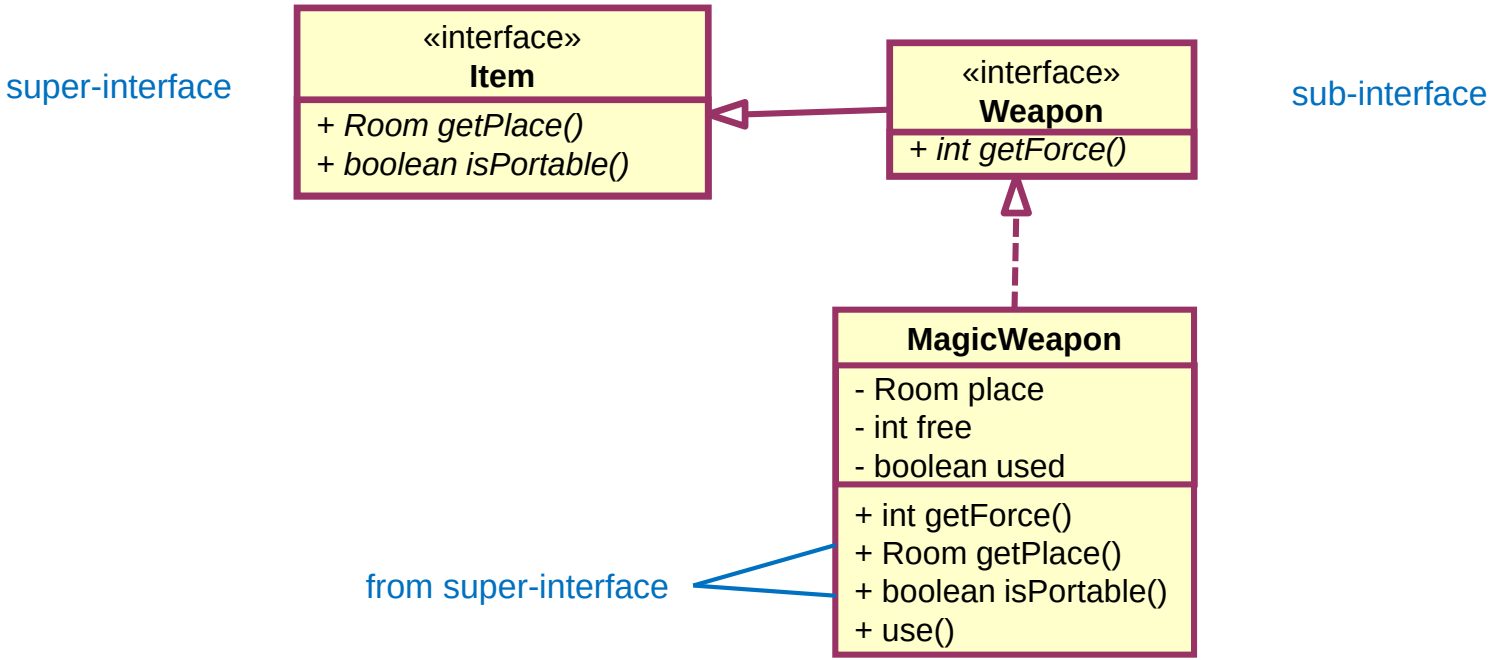
```
interface Comparable {
    /**
     * @return negative, zero, or positive if this
     *   object is less than, equal to, or greater than o
     */
    int compareTo(Object o);
}
```

- Implemented by `java.lang.String`: alphabetical order
  - `class String implements Comparable`
  - What is the result of `"this".compareTo("that")`?
- Implemented by `java.util.Date`: temporal order
  - `new Date(2013,11,26).compareTo(new Date(2014,11,26))`?

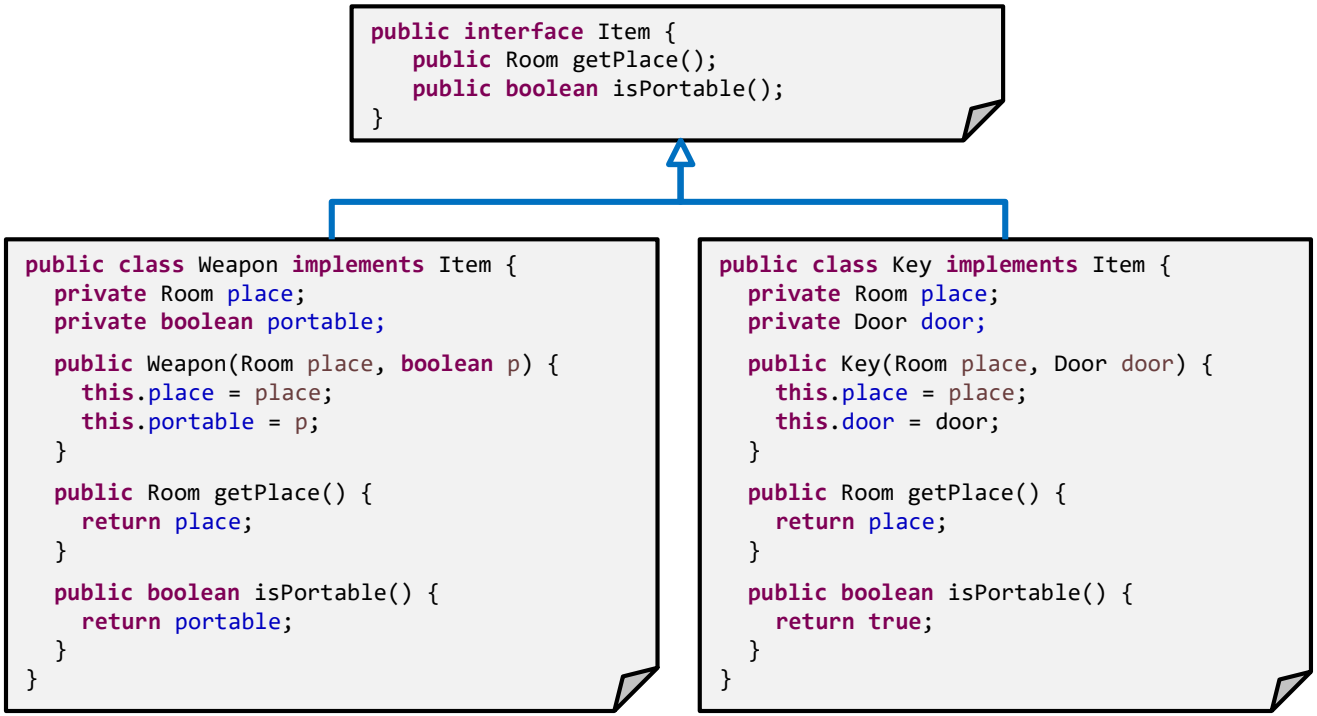
NOTATION


«interface»
Item
+ Room getPlace()
+ boolean isPortable()
Item
- Room place
+ Room getPlace()
+ boolean isPortable()
Notation is slightly different!
Key
- Room place
- Door door
+ Room getPlace()
+ boolean isPortable
+ boolen opens(Door door)
Key
- Door door
+ boolean isPortable()
+ boolean opens(Door door)

COMBINING INTERFACES AND INHERITANCE



ABSTRACT CLASSES: EXAMPLE

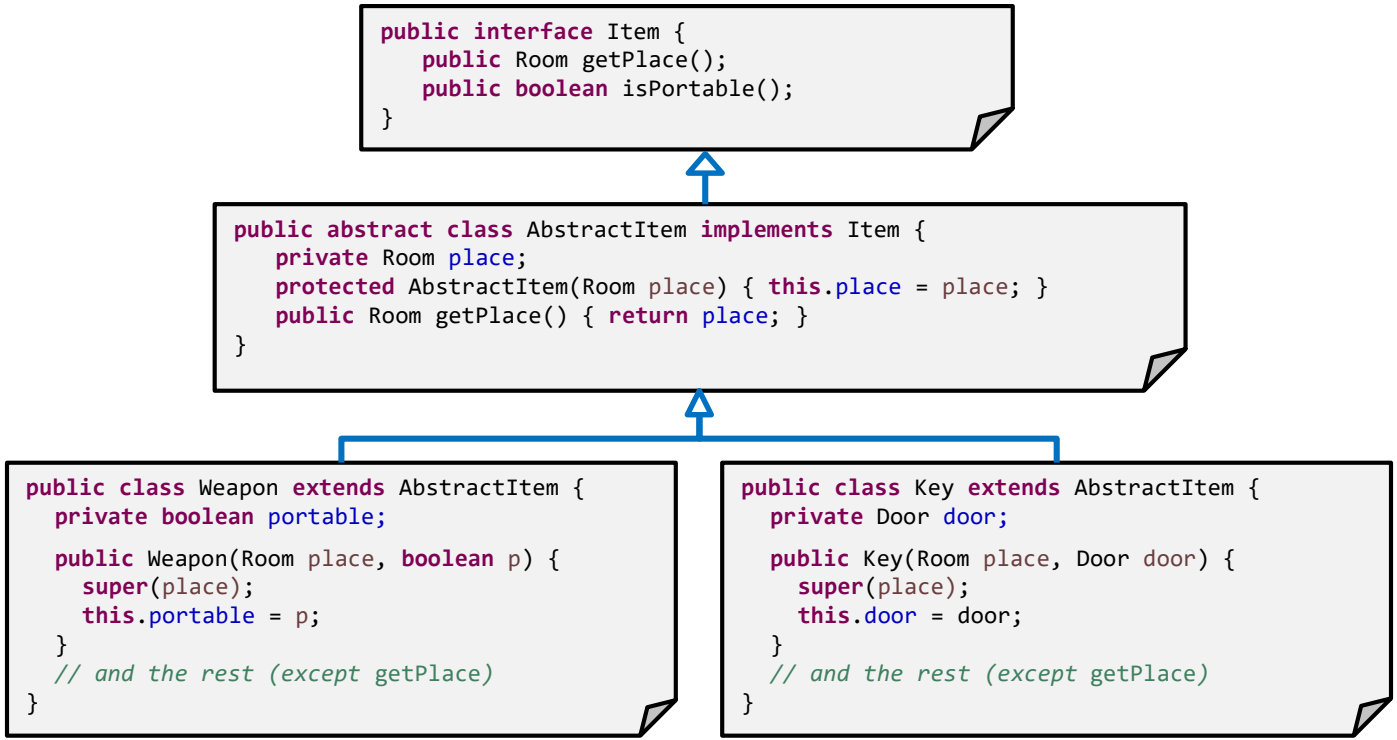


```
public interface Item {
    public Room getPlace();
    public boolean isPortable();
}
```

```
public class Weapon implements Item {
  private Room place;
  private boolean portable;

  public Weapon(Room place, boolean p) {
    this.place = place;
    this.portable = p;
  }

  public Room getPlace() {
    return place;
  }

  public boolean isPortable() {
    return portable;
  }
}
```

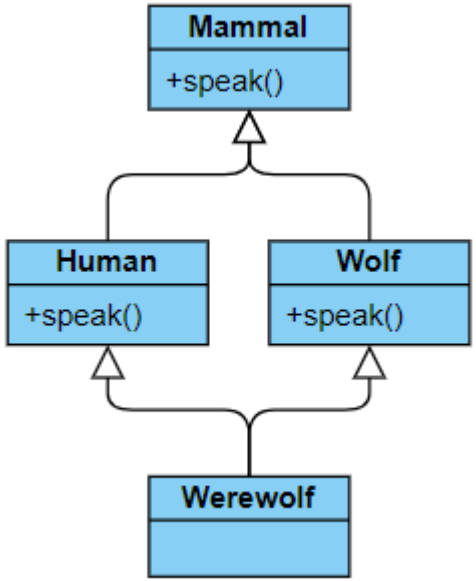
```
public class Key implements Item {
  private Room place;
  private Door door;

  public Key(Room place, Door door) {
    this.place = place;
    this.door = door;
  }

  public Room getPlace() {
    return place;
  }

  public boolean isPortable() {
    return true;
  }
}
```

ABSTRACT CLASSES: EXAMPLE (CONTINUED)



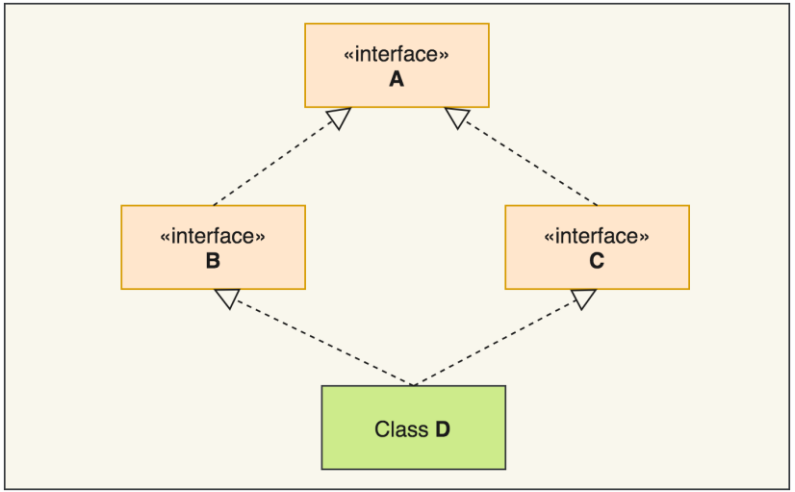
MULTIPLE INHERITANCE

- In many programming languages, a class can extend multiple classes
- This leads to the famous diamond problem



MULTIPLE INHERITANCE IN JAVA

- In Java, classes can implement multiple interfaces



- However, interfaces have no method body so no problem!

DEFAULT METHODS IN INTERFACES

Java 8 introduces static methods and default methods to interfaces

```
interface A {
   public int getANumber();
}
```

```
public class B implements A {
   private int theBestNumberFolks;

   public int getANumber() {
       return theBestNumberFolks;
   }
}
```

```
interface A {
   default public int getANumber() {
      return 42;
   }
}
```

…. oops? The diamond problem is back!

DEFAULT METHODS IN INTERFACES

Conflict resolution (= which same-signature method does the class inherit??)

1. Method inherited from a (super)class take priority over default interface methods
2. Methods from subinterfaces take priority over the superinterfaces
3. …? Error! The implementing class must provide its own implementation!!

BENEFITS OF INTERFACES OVER INHERITANCE

- No default implementation of methods necessary
- Classes can implement multiple interfaces
  - Possibly combined with extending one class
  - Default methods are inherited from each interface

```
public class Wand implements Weapon, Magical extends WoodenItem {
    // class body
}
```

BENEFITS OF INHERITANCE OVER INTERFACES

- More reuse of methods (default methods in interfaces are limited)
- Classes and abstract classes can have non-final fields
- Classes can have protected and private members
- Abstract classes are a “basis for subclasses with shared behaviour”
- Interfaces are specifications, describing the behaviour an implementing class will have

INHERITANCE VS COMPOSITION

- Inheritance: inherit fields and methods from another class
  - Use when there is a clear parent-child relationship of concepts (is-a relationship)
  - Use to alter the behaviour of a class
  - Use when you want to reuse the entire interface of the superclass
- Composition: rely on other object(s) to provide (some) functionality
  - Composition is often more appropriate
  - Use when only using parts of the functionality of another class (has-a or uses-a relationship)
- Both are fundamental in object-oriented programming!

THE OBJECT CLASS

- All classes are derived from the base class **Object** (except primitive types)
- All classes inherit its methods, for example:

```
public boolean equals(Object o)  // check for “equality”
public int hashCode()            // compute unique representation (next week)
public String toString()         // create a String representation
```

- These methods have a default implementation.
- Often it is a good idea to override these inherited methods
  - The default `toString()` method returns a `String` that represents the internal reference to the instance: “`SomeObjectClassname@hashcodenumber`”
  - Usually not very useful, so replace it with a better `toString()` implementation!

INHERITING FROM OBJECT: EXAMPLE

- What does the following print?

```
1 Point2D p = new Point2D(2, 3);
2 System.out.println("p = " + p);
3 System.out.println(p == new Point2D(2, 3));
4 System.out.println(p.equals(new Point2D(2, 3)));
5 System.out.println(p.equals(new Point3D(2, 3, 4)));
```

system-generated object ID

```
1 p = Point2D@70dea4e
2 false
3 false
4 false
```

- How does this change if `Object` methods are overridden?

```
public class Point2D {
   // ... (as before)
   public String toString() {
      return "Point at " + getX() + "," + getY();
   }
   public boolean equals(Object obj) {
      if (!(obj instanceof Point2D)) { return false; }
      Point2D other = (Point2D) obj;
      return other.getX() == getX() && other.getY() == getY();
   }
}
```

```
1 p = Point at 2,3
  (toString invoked)
2 false
  (still different objects!)
3 true
  (equals invoked)
4 true
  (equals does not notice
  that obj is actually Point3D)
```

SUMMARY SO FAR

- Subclass, superclass, abstract class, interface
- A class can extend at most one class and implement multiple interfaces
- An interface can extend multiple interfaces
- Abstract methods have no method body
- Abstract methods live in abstract classes
- Inheritance, multiple inheritance and the diamond problem
- Overriding, overloading, polymorphism
- The `Object` class

TYPES REVISITED

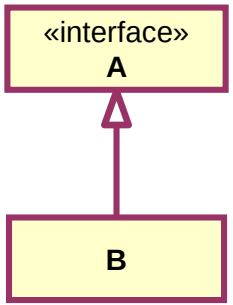
- Java has two kinds of data types
  - Primitive types (int, boolean, double, etc.)
  - Reference types (classes and interfaces, for example String, List, etc)
- Reference types carry a subtyping relation
  - One type can be a subtype of another (fundamental concept)
  - Subtyping is a partial order (meaning reflexive and transitive)
    - Reflexive: every type is a subtype of itself: `A` is a subtype of `A`
    - Transitive: if `A` is a subtype of `B` and `B` is a subtype of `C`, then `A` is a subtype of `C`
- Subtyping in programming language theory: substitutability
  - if S is a subtype of T, then we can safely use S when T is expected
  - (Whenever a value of a given type is expected, a subtype can be used)

SUBTYPING IN JAVA

In Java, when is `A` a subtype of `B`? If `A` extends or implements `B`

class B **implements** A

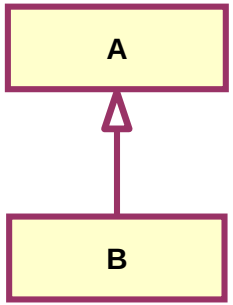
supertype



subtype

class B **extends** A

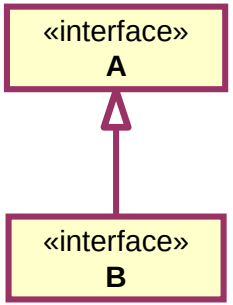
supertype



subtype

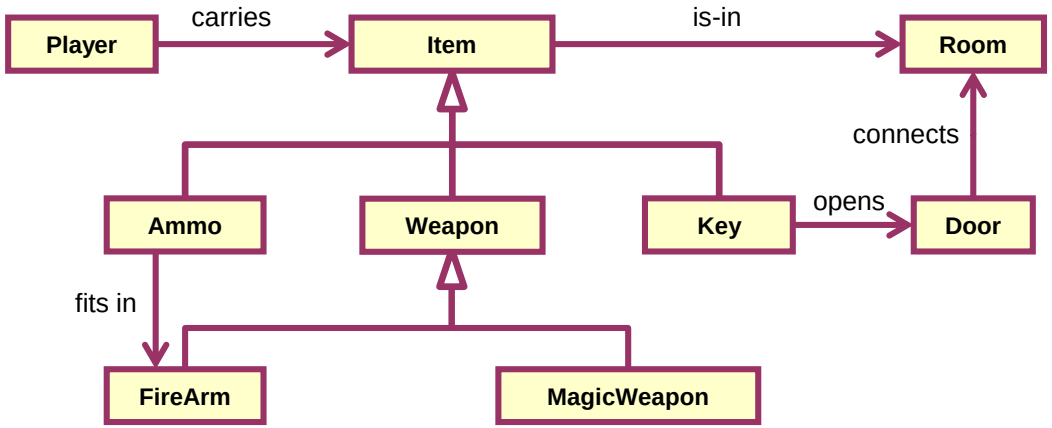
interface B **extends** A

supertype



subtype

EXAMPLE



- Subtypes of `Weapon`?
  - `FireArm`, `MagicWeapon`, `Weapon`
- Subtypes of `Item`?
  - `Ammo`, `Weapon`, `Key`, `FireArm`, `MagicWeapon`, `Item`
- Supertypes of `Key`?
  - `Key`, `Item`, `Object`

SUBTYPE IS ALLOWED WHERE SUPERTYPE IS EXPECTED

- Where does the following program fragment go wrong?

```
Key k1 = new Key();
System.out.println(k1.isPortable());
System.out.println(k1.opens(door));
Item i1 = k1;
System.out.println(i1.isPortable());
System.out.println(i1.opens(door));
Key k2 = i1;
```

`isPortable()` is inherited method of Item
`opens(Door d)` is method of `Key`
subtype value assigned to supertype
`isPortable()` can also be called on `i1`
`opens(Door d)` can *not* be called on `i1`
supertype can *not* be assigned to subtype

- How can this be? i1 and k1 are the same object!

STATIC VERSUS DYNAMIC TYPE OF AN EXPRESSION

- **Static type**: that which the compiler can infer during “compile-time”
  - Also called *declared type*
  - `i1` has static type `Item` (that’s how it was declared)
  - The Java compiler will **not** do (potentially complicated) type inferencing: if you **declare** i1 to be an “Item”, it will be treated as an “Item”, *even* when “everyone can see” that it *actually* will contain a “Key” at run time.
- **Dynamic type**: that which the value actually has during “run-time”
  - Also called *actual type* or *run-time type*
  - `i1` has dynamic type `Key` (because `k1` was assigned to it)
  - At some other point, `i1` may have dynamic type `Item.`
- The dynamic type is always a subtype of the static type
  - What the compiler considers correct is based on the static type.

DYNAMIC TYPE TEST

- How to find out the dynamic type of an expression `expr` during “run time”?
  - Type test: `expr instanceof Type`
  - This yields **`true`** if the dynamic type of `expr` is a *subtype* of `Type`
  - `null instanceof Type` is always “**false**”.
- What does the following print?

```
Key k1 = new Key();
Item i1 = k1;
Item i2 = new FireArm();
Item i3 = null;
System.out.println(k1 instanceof Key);
System.out.println(k1 instanceof Item);
System.out.println(i1 instanceof Key);
System.out.println(i1 instanceof Item);
System.out.println(i2 instanceof Key);
System.out.println(i2 instanceof Item);
System.out.println(i3 instanceof Item);
```

```
true
true
true
true
false
true
false
```

STATIC TYPE CHANGE (CAST)

- How to tell the compiler the actual type of an expression `expr`?
  - **Why** would you even want that? Because you can then call appropriate methods on it, that are not available for the super type.
  - How? Type cast: `(Type) expr` changes the static type to `Type`
  - Only correct if *dynamic* type of `expr` is a subtype of `Type`
    - *You cannot change the dynamic type of an expression*

```
Key k1 = new Key();
Item i1 = k1;
System.out.println(i1.isPortable());
System.out.println(((Key) i1).opens(door));
Key k2 = (Key) i1;
```

correct because dynamic type of `i1` is actually `Key` here

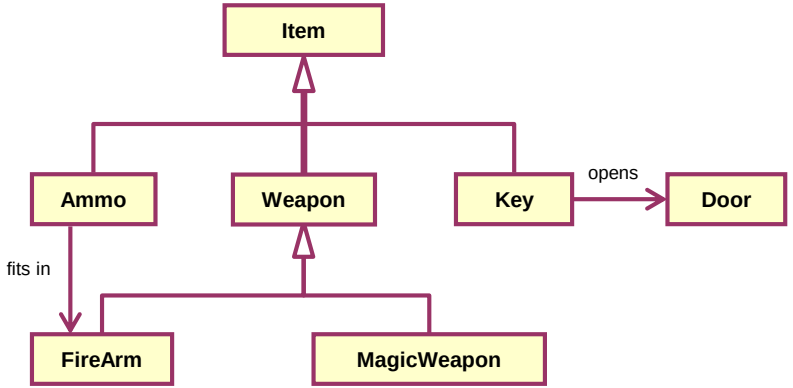
- Watch the parentheses
  - `((Key) i1).opens(door)` is correct: `((Key) i1)` has type "Key".
  - `(Key) i1.opens(door)` is wrong: tries to cast the result from the method.

EXAMPLE

```java
public class Player {
    private Item item;
    // other code
    /** Tests if item is a Key. */
    public boolean hasKey() {
        return item instanceof Key;
    }
    /** Returns the item if it is a key, otherwise null. */
    public Key getKey() {
        return item instanceof Key ? (Key) item : null;
    }
    /** Fires the firearm, if item is a firearm. */
    public boolean fire() {
        return item instanceof FireArm && ((FireArm) item).fire();
    }
}
```

```java
public class FireArm implements Weapon {
    private Ammo ammo;
    public boolean fire() {
        boolean result = ammo != null;
        ammo = null;
        return result;
    }
}
```

TEST




```
Item i1 = new FireArm();
Key k1 = (Key) i1;
Item i2 = null;
Key k2 = (Key) i2;
k2.opens(door);
((FireArm) i1).fire();
FireArm f1 = i1;
FireArm f2 = (Weapon) i1;
FireArm f3 = new Item();
FireArm f4 = (FireArm) new Weapon();
Weapon w1 = (Item) new FireArm();
Weapon w2 = (Weapon) new FireArm();
Weapon w3 = new MagicWeapon();
boolean b1 = w2 instanceof Item;
boolean b2 = w2 instanceof Object;
boolean b3 = w2 instanceof FireArm;
```

Correct: `FireArm` is subtype of Item
Wrong: dynamic type of `i1` is not a subtype of `Key`
Correct: **`null`** is a value of all reference types
Correct: **`null`** is a value of all reference types
Wrong: `k2` is **`null`**
Correct: dynamic type of `i1` is `FireArm`
Wrong: static type of `i1` is Item, not `FireArm`
Wrong: static type of `(Weapon) i1` is `Weapon`
Wrong: static (& dynamic) type of **`new`** `Item()` is `Item`
Wrong: dynamic type of **`new`** `Weapon()` is `Weapon`
Wrong: static type of `(Item)` **`new`** `FireArm()` is `Item`
Correct but cast is unnecessary
Correct: `MagicWeapon` is a subtype of `Weapon`
`b1` becomes **`true`**
`b2` becomes **`true`**
`b3` becomes **`true`**

TAKE HOME MESSAGES

- The class hierarchy of classes that extend and implement
- Abstract classes and interfaces
- Multiple inheritance and the diamond problem
- Polymorphism, both static (overloading) and dynamic (overriding)
- Subtyping and type casting
- When to use interfaces or abstract classes or composition