Programming Paradigms
159.272
Encapsulation

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Java Tutorial: Controlling Access to Members of a Class
   http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Overview

- information hiding
- access modifiers
- fields vs properties
- setters and getters
- lazy initialisation
- hiding classes: non public and inner classes
- ownership
- advanced encapsulation with class loaders

Information Hiding

- object-oriented programs are composed by connecting simple building blocks (objects created from classes)
- objects can have behaviour that can be accessed through a simple interface, but is in itself rather complex

Information Hiding With Interfaces

```
Comparable[] comparable = ...;
Arrays.sort(comparable);
```

- the actual comparable objects can be complex
- in particular, the `compare` method can be sophisticated
- e.g., the objects can be instances of complex class like `Student` : `compare()` compares first by `dateOfBirth`, then by `name`, then by `firstName`
- `Arrays.sort` does not see this complexity

Objects as Blackboxes

- we want to treat objects as **blackboxes**
- objects have visible features (state and behaviour)
- but we do **not** have to know how this is achieved
- there might be invisible features supporting visible features

Example from the Physical World: USB

a simple interface,
to use it, only this is what we **need to know about**

a complex implementation, but **we don't have to know about this to use it**

Dealing with Long and Complex Methods

- **long** and **complex** methods are difficult to understand and to maintain
- complex means: many (nested) control structures like loops and conditionals
- long means: many lines of code
- it is better to have multiple shorter methods
- some methods are "helper" methods

A Long and Complex Method

```
// parse data in an array into instances of Student
public Student[] parse(String[][] data) {
          Student[] students = new Student[data.length];
          for (int i=0;i<students.length;i++) {
                    String[] row = data[i];
                    if (row.length>3) {
                              System.err.println("Too many values in row " + i);
                    }
                    else if  (row.length<3) {
                              System.err.println("Not enough values in row " + i);
                    }
                    else {
                              Student student = new Student();
                              student.id = Integer.parseInt(row[0]);
                              student.firstName = row[1];
                              student.lastName = row[2];
                              students[i] = student;
                    }
          }
          return students;}
```

Splitting a Long Method

```
// parse data in an array into instances of Student
public Student[] parse(String[][] data) {
          Student[] students = new Student[data.length];
          for (int i=0;i<students.length;i++) {
                    String[] row = data[i];
                    if (row.length>3) {
                              System.err.println("Too many valu
                    }
                    else if  (row.length<3) {
                              System.err.println("Not enough values
                    }
                    else {
                              Student student = new Student();
                              student.id = Integer.parseInt(row[0]);
                              student.firstName = row[1];
                              student.lastName = row[2];
                              students[i] = student;
                    }
          }
          return students;}
```

move this into a new method

Splitting a Long Method ctd

```
// parse data in an array into instances of Student
public Student[] parse(String[][] data) {
        Student[] students = new Student[data.length];
        for (int i=0;i<students.length;i++) {
                String[] row = data[i];
                Student student = this.parseRow(row,i);
                if (student!=null) {
                        students[i] = student;
                }
        }
        return students;
}
```

Splitting A Long Method ctd

```
// parse a single row only
public Student parseRow(String[] row,int rowNo) {
        if (row.length>3) {
                System.err.println("To many values in row " + rowNo);
        }
        else if  (row.length<3) {
                System.err.println("Not enough values in row " + rowNo);
        }
        else {
                Student student = new Student();
                student.id = Integer.parseInt(row[0]);
                student.firstName = row[1];
                student.lastName = row[2];
                return student;
        }
        return null;
}
```

Encapsulation

- splitting longer methods makes them easier to comprehend, test and to maintain
- the main `parse` method uses an internal helper method `parseRow` to parse a single row
- the sole purpose of this method is to support `parse`
- if `parse` is modified, `parseRow` could be changed as well, or could even be deleted
- therefore, we don't want this method to be used from other methods
- this can be achieved with **encapsulation**

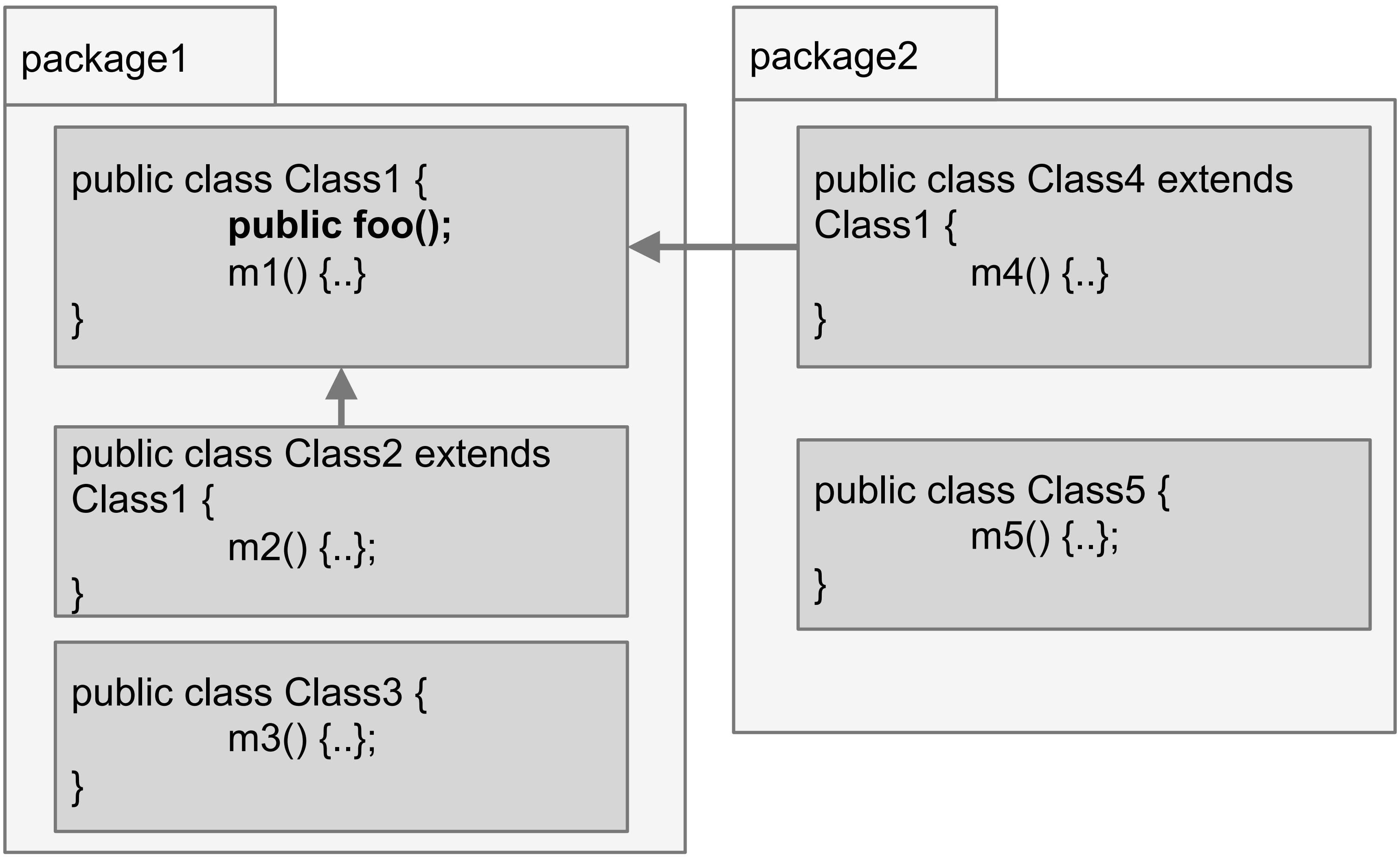
Visibility

- it is not possible to enforce that `parseRow` can only be used by `parse`
- but it is possible to enforce that `parseRow` can only be used by other features (methods, constructors, fields or static blocks) **within the same class**: by declaring the method as `private`
- use = access
  - use in methods, constructors and static blocks: invocation
  - use in fields: invocation in field initialiser
- `public` and `private` are **access modifiers**
- they are used to declare **access rules** (aka **visibility rules**)
- these rules are enforced by the compiler

Access Modifiers and Visibility Rules

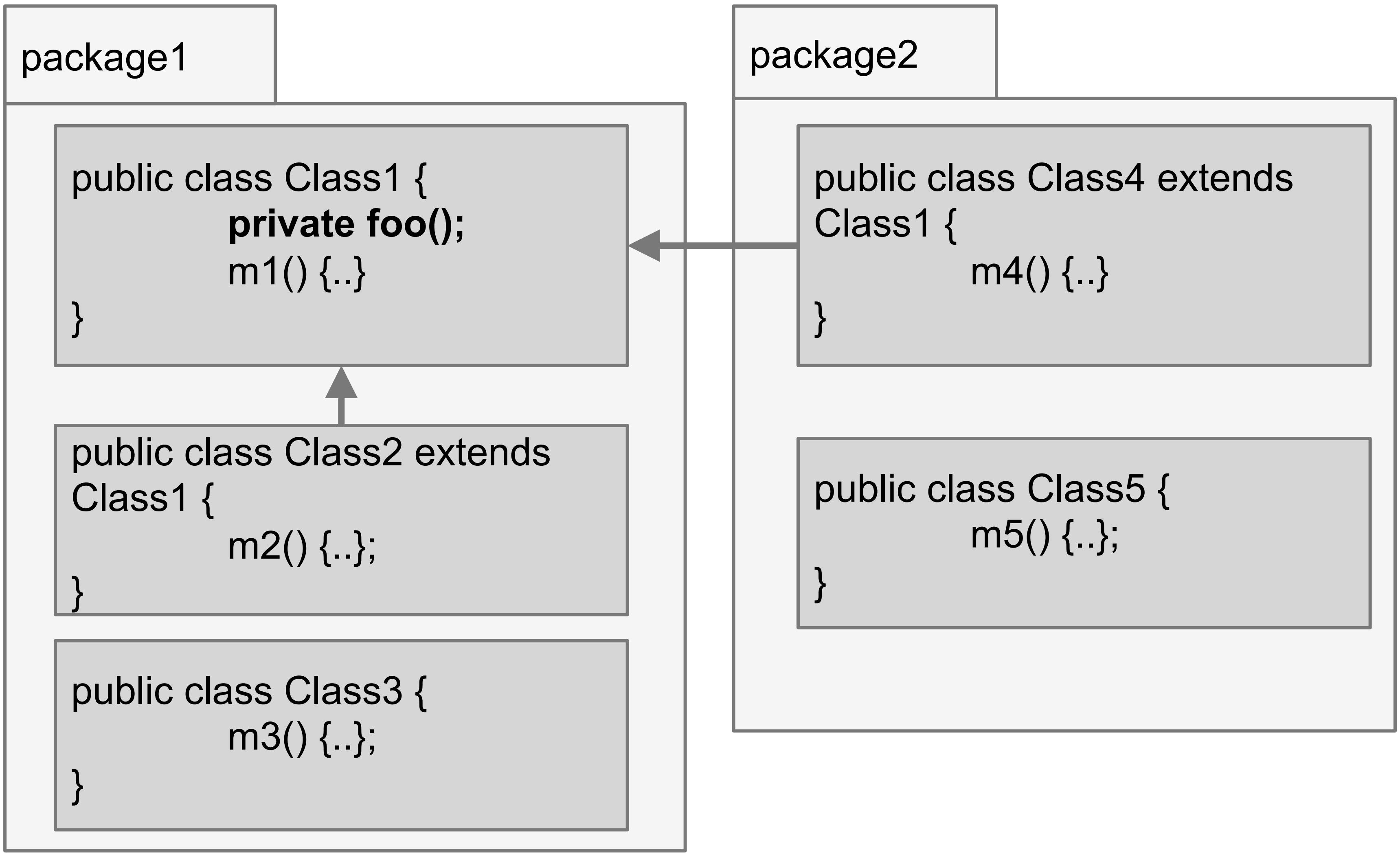
- `public` methods and fields can be accessed from all other classes
- `protected` methods and fields can only be accessed from classes within the same package, or subclasses
- `private` methods and fields can only be accessed from the class that defines the respective method or field
- default methods and fields (no explicit access modifier) can only be accessed from classes within the same package
- default visibility is also called **package-private**

Which Methods Can Access `public foo()` ?

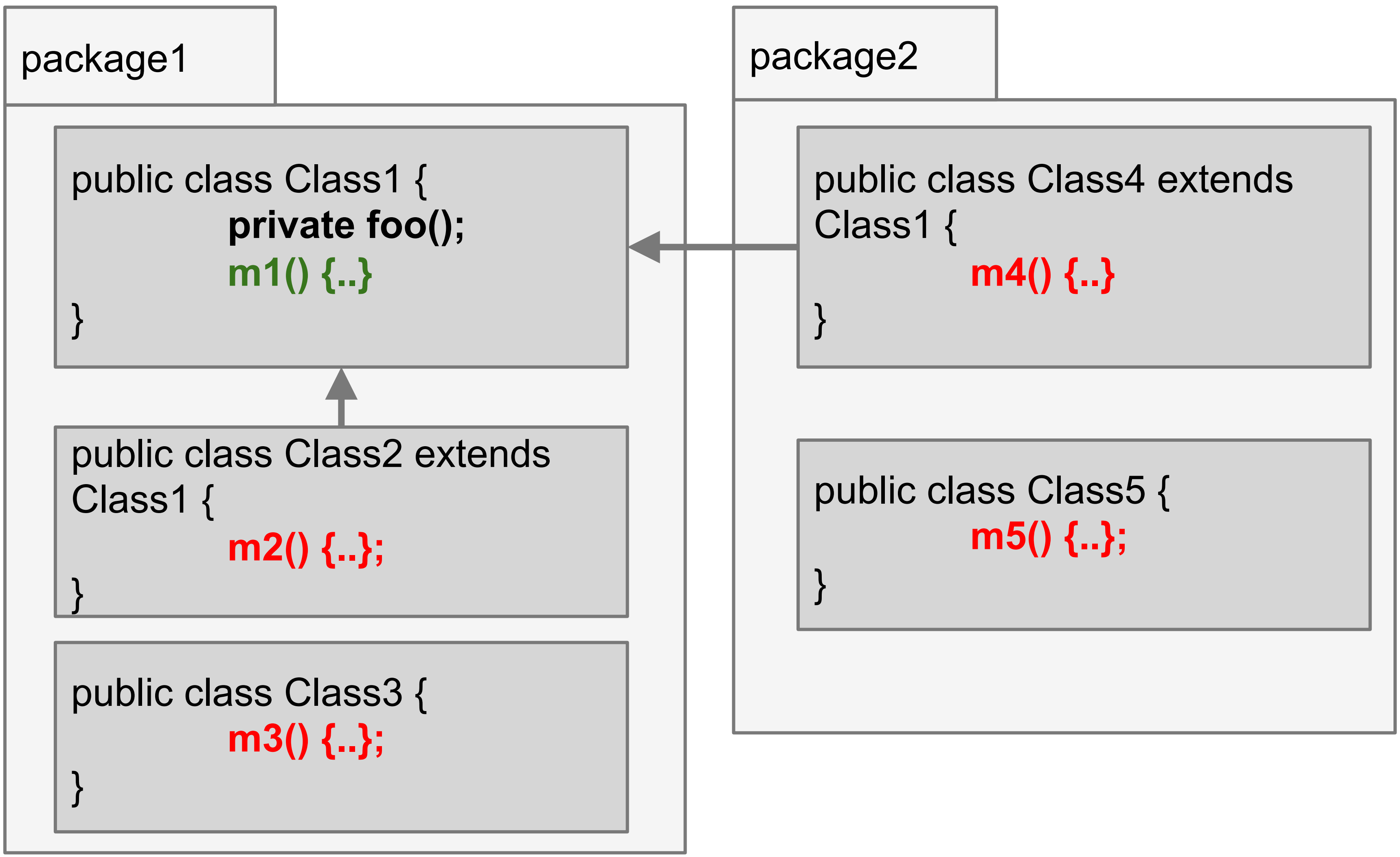


Which Methods Can Access `public foo()` ?

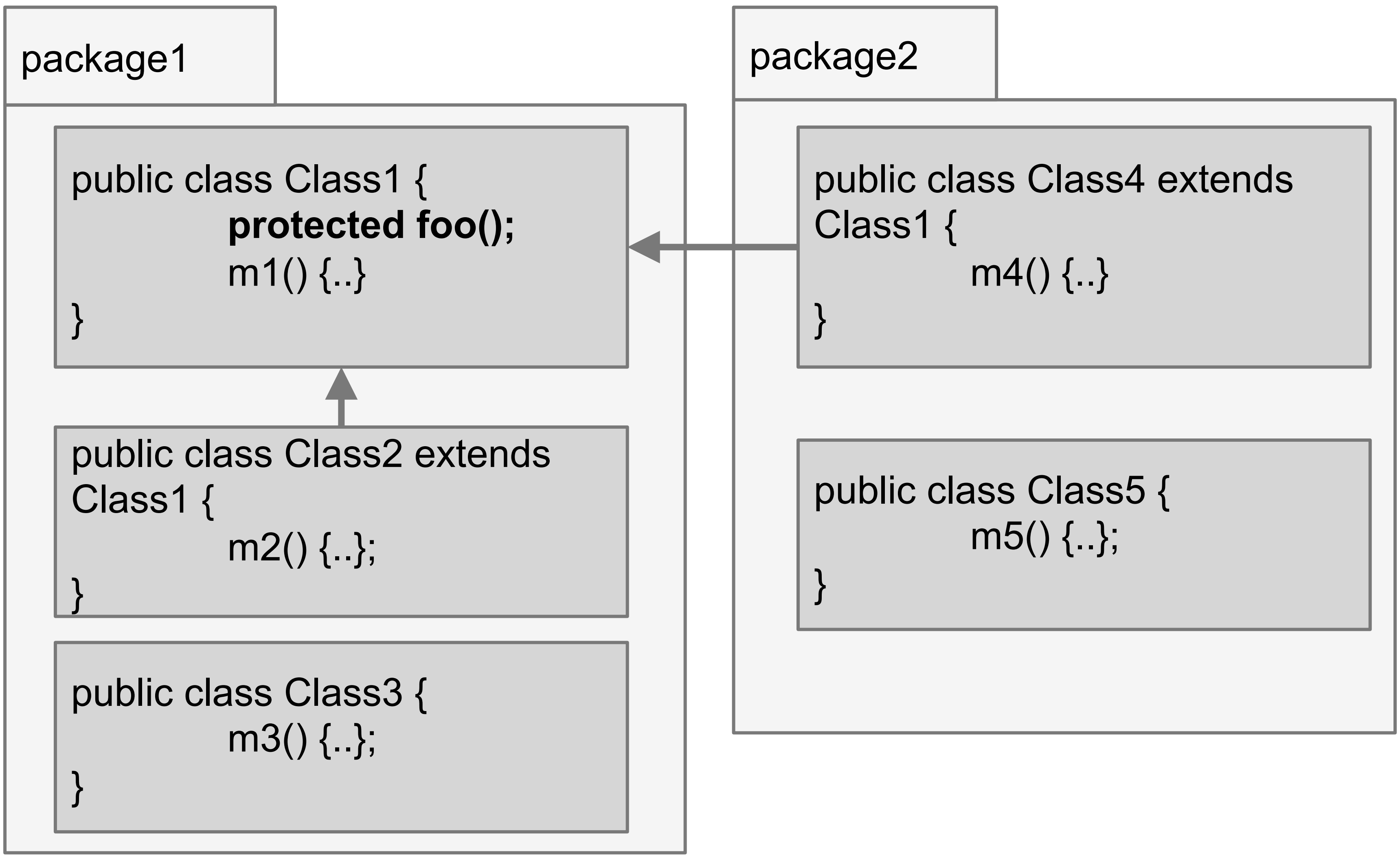
Which Methods Can Access `private foo()` ?



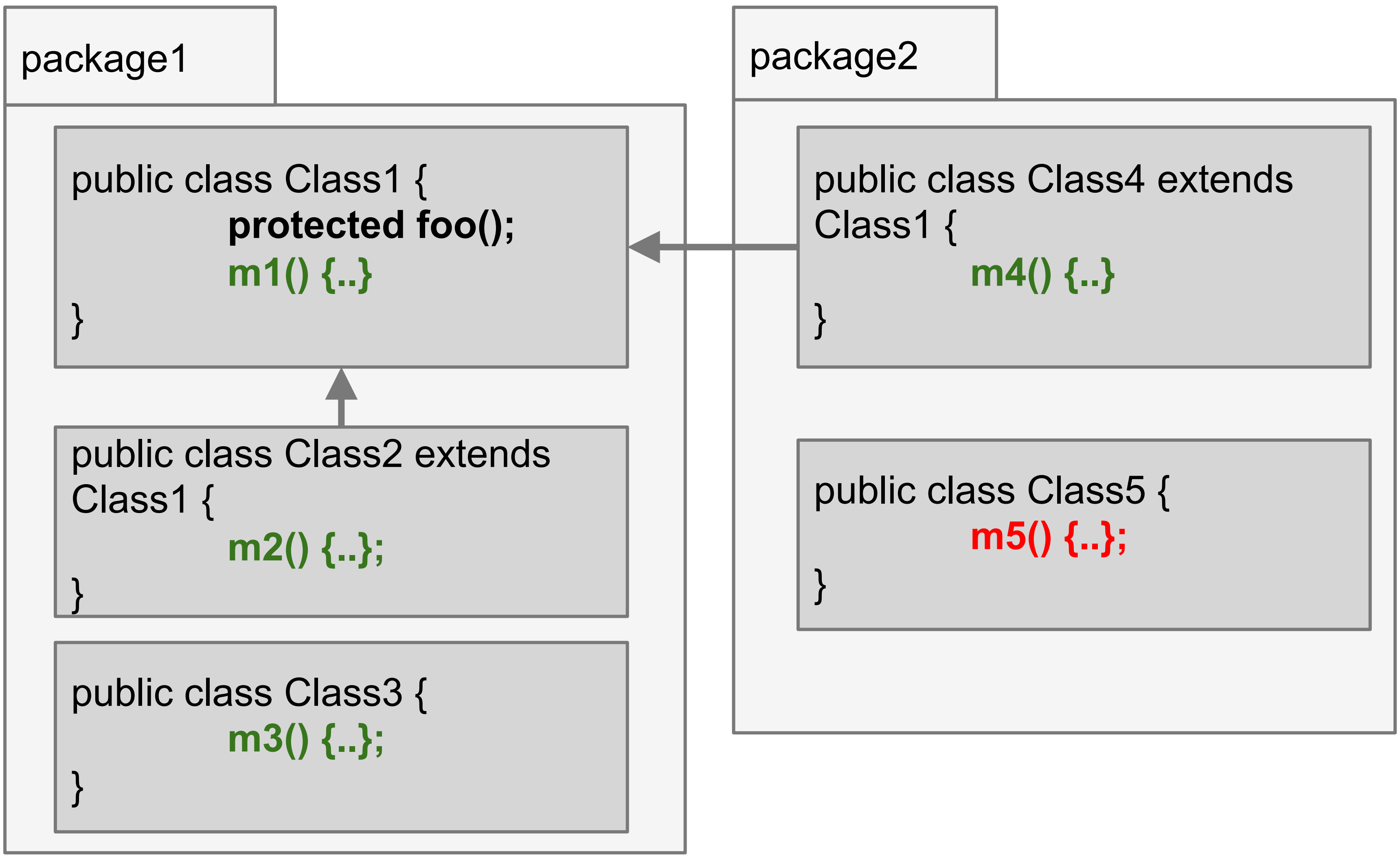
Which Methods Can Access `private foo()` ?



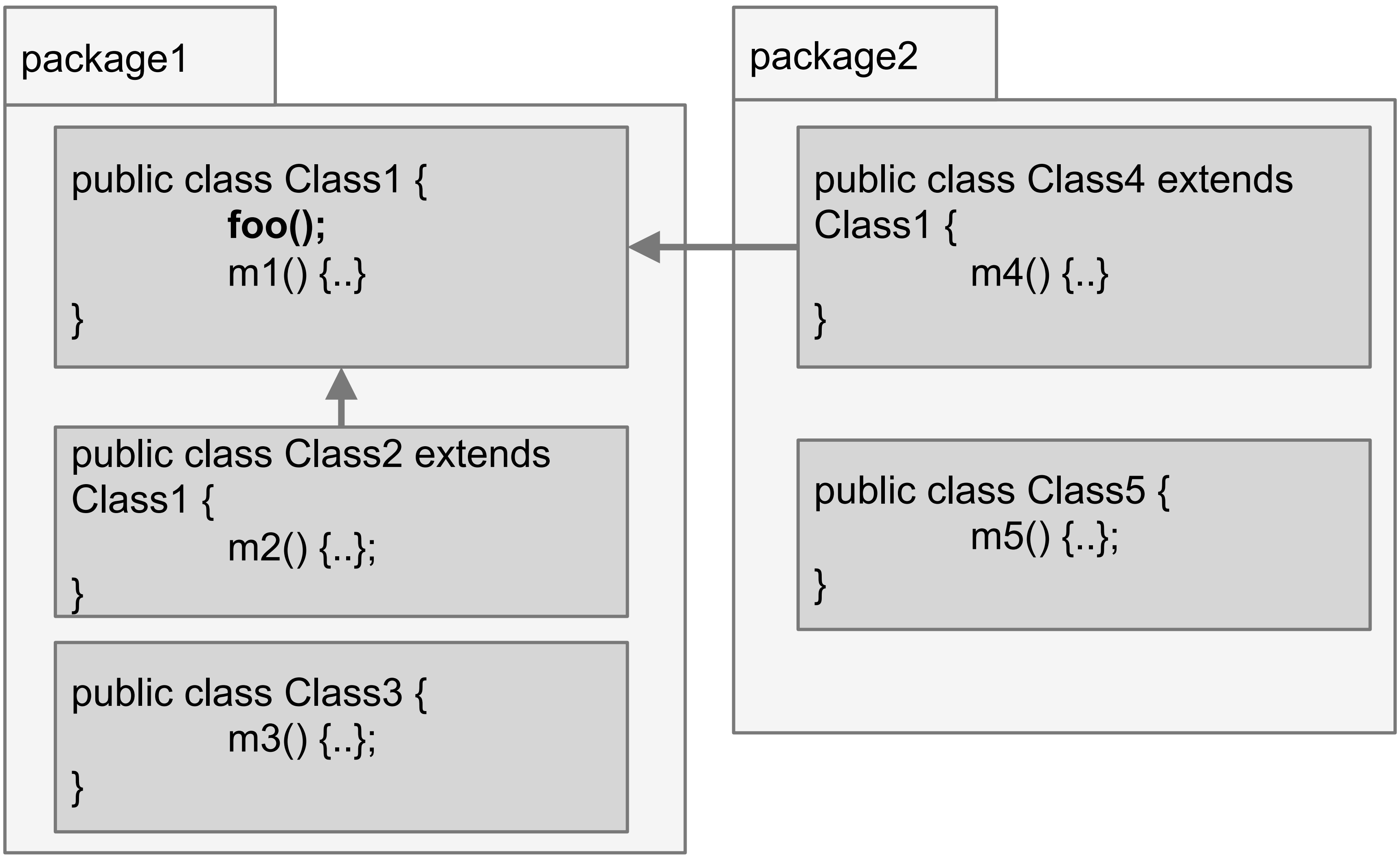
Which Methods Can Access `protected foo()` ?



Which Methods Can Access `protected foo()` ?



Which Methods Can Access `foo()`?



Which Methods Can Access `foo()` ?

Encapsulating State

- it is widely considered as good design to hide state
- this is usually done by declaring instance variables as private (or sometimes protected)
- access to instance variables is then possible through:
    - dedicated public read methods (aka **getter** or **accessors**)
    - dedicated public write methods (aka **setters** or **mutators**)
    - constructors

Getters and Setters

- **naming conventions** are used in Java for getters and setters
- getters (accessors):
  - use the `get` prefix (if the type is `boolean`, the prefix `is` can be used as well), examples: `getName()`, `isParttimeStudent()`
  - signature: no parameter, returns the field type
- setters (mutators):
  - use the `set` prefix, example: `setName(String)`
  - signature: return type is void, single parameter, type is field type
- accessors and mutators are public
- the fields are private (or sometimes protected)

Getters and Setters Example

```
public class Student {
      private int id = 0;

   public void setId(int id) {
            this.id = id;
   }

   public int getId() {
      return this.id;
   }
}
```

private instance variables

public setter

note that this.id references the field, while id is the method parameter

public getter

Getters and Setters ctd

- the combination of a getter and a setter is also called a **property**
- properties are defined in the **JavaBean** specification - a simple component model for Java
- it is good practise to use properties (instead of accessing public fields directly) - many tools rely on this, including:
  - XML based serialisation (storing) of object graphs
  - graphical user interface builders
- because getters and setters use a fixed pattern, they can be easily generated by tools
  (in Eclipse: Source > Generate Getters and Setters)
- still getters and setters create some overhead - so what is the advantage?

Getter Use Case

```
public class Student {
        private int id = 42;
        ..
    private String enrollmentStatus =
        StudentDB.getStatus(this.id);
        public String getEnrollmentStatus() {
        return this.enrollmentStatus;
    }
        ..
}
```

- getting the enrollment status is expensive: it invokes a call to `StudentDB.getStatus()`, and this may result in a network call
- this must be done whenever `Student` is instantiated !

Getter Use Case ctd

```
public class Student {
        private int id = 42;
        ..
    private String enrollmentStatus=null;
        public String getEnrollmentStatus() {
                if (this.enrollmentStatus==null) {

    this.enrollmentStatus=StudentDB.getStatus(this.id);
        }
        return this.enrollmentStatus;
    }
        ..
}
```

- now the enrollment status is only initialised when it is actually needed (i.e. when the `getter` is invoked).

Lazy Initialisation

- this technique is called **lazy initialisation**, as opposed to **eager initialisation** (initialisation in the constructor, or directly in the field definition)
- note that this is transparent (invisible) to the programmer as other classes have no direct access to the field

Setter Use Case

```
import java.util.Date;
public class Student {
        private Date dob = null;
        ..
        public void setDob(Date dob) {
                int year = dob.getYear() + 1900;
                if (year < 1913) {
                        System.out.println("too old to study");
        }
        else {
            this.dob = dob;
        }
    }
        ..
}
```

only set value if it passes some **validation checks**

Visibility of Constructors

- constructors can have access modifiers as well
- use case: the **singleton** pattern
- singleton is an example of a **design pattern**
- design patterns are reusable object design fragments that are language independent
- problem solved by singleton: write a class that can only have a single instance
- use case: "global" objects like DataBase or System *(note that those are not necessarily actual classes)*

The Singleton Pattern

```
public class DataBase {
        public static DataBase soleInstance = new DataBase();
        private DataBase () {
                super();
    }
        .. //
}
```

here we can access the private constructor !

one instance is exposed (public)

because the constructor is private, no instance can be created from outside this class

`DataBase.soleInstance` can be used to reference the only instance of `DataBase` that can be created

Visibility of Classes

- classes can have either public or default visibility
- it is also possible to define classes within classes (inner classes)
- inner classes can be declared as `private`, `protected`, package private (default) or `public`
- inner classes have access to private features of the enclosing class

Access Rules and Inheritance

```
public class Mammal {
       public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
       @Override
       protected boolean hasStripes() {return true;}
}
```

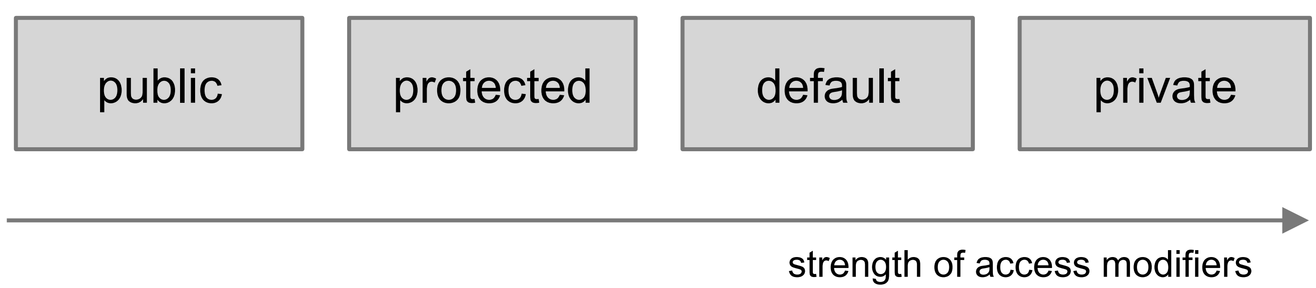
Access Rules and Inheritance

```
public class Mammal {
	public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
	@Override
	protected boolean hasStripes() {return true;}
}
```

- this should (and will) fail
- the compiler rejects this
- references to `hasStripes()` from classes in other packages that are not subclasses of `Zebra` (i.e., classes relying on `hasStripes()` being `public`) would not be able to access the overridden method

Access Rules and Inheritance

- the general rule is that when overriding methods, the visibility can be **relaxed**, but **not strengthened**
- this is part of a more general rule (Liskov's Substitution Principle LSP): guarantees made by a method (such as visibility rules) cannot be weakened in subclasses



The Limitations of Encapsulation With Access Modifiers

- even when using the private access modifier, there are limitations
- private fields are not protected from access from other objects within the same class
- the objects referenced in private fields are not protected from modification by other objects referencing them
- this means that encapsulation with private **does not imply ownership**
- we discuss some related examples next
- example source code:

  http://oop-examples.googlecode.com/svn/mutability/

Example: the EvilTwin

```
public class A {
        private int x = 0;
        private A master = null;
        public A(int x) {
                super();
                this.x = x;
        }
        public int getX() {return x;}
        public void setX(int x) {
                this.x = x;
                if (this.master!=null) {this.master.x = x;}
        }
        public A clone() {
                A clone = new A(this.x);
                clone.master = this;
                return clone;
        }
}
```

access to a private field of another object !!

Example: the Evil Twin ctd

```
A a = new A(42);
System.out.println("a.x = " + a.getX());
A b = a.clone();
b.setX(43);
System.out.println("a.x = " + a.getX());
```

Example: the Evil Twin ctd

```
A a = new A(42);
System.out.println("a.x = " + a.getX());
A b = a.clone();
b.setX(43);
System.out.println("a.x = " + a.getX());
```

- this will first print 42, then 43
- `private` will not protect a field from access from arbitrary other objects, only from access from objects which do not instantiate this class !

Advanced Encapsulation With Class Loaders

- the encapsulation mechanisms built into Java are sometimes not sufficient to address the requirements in large-scale (enterprise) applications
- but is is possible to control access to classes and entire packages using **classloaders**
- with classloaders access to entire packages can be controlled
- this can be used to support **modules** that export and import entire packages
- an example where this is used is OSGi
- a detailed discussion is outside the scope of this paper, but this is part of 300/400 level SE papers