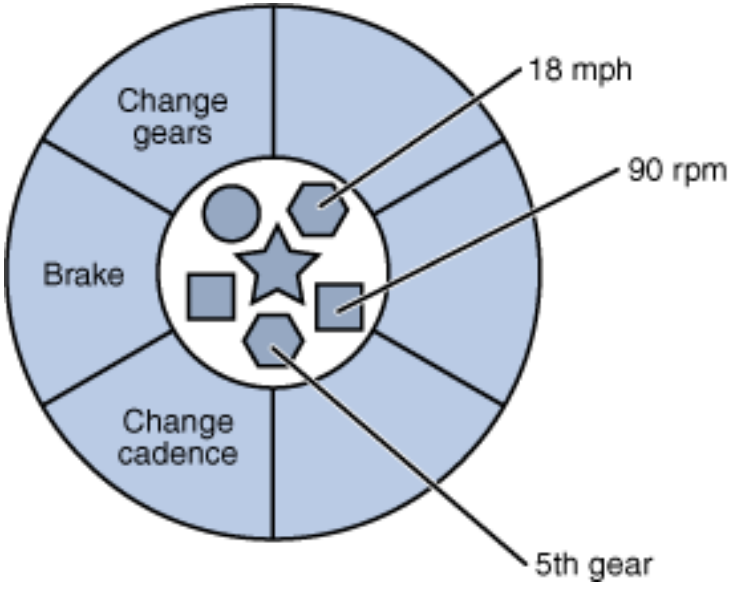
Software and Programming II

Objects and Classes, and Inheritance, and ...

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk

October 7, 2014



Classes

- A class describes a set of objects
- The objects are called instances of the class
- A class describes:

  Fields (instance variables)that hold the data for each object

  Constructors that tell how to create a new object of this class

  Methods that describe the actions the object can perform
- In addition, a class can have data and methods of its own (not part of the objects)
  - For example, it can keep a count of the number of objects it has created
  - Such data and methods are called static

Defining a class

- Here is the simplest syntax for defining a class:

```
class ClassName {
    // the fields (variables) of the object
    // the constructors for the object
    // the methods of the object
}
```

- You can put `public`, `protected`, or `private` before the word `class`
- Things in a class can be in any order (we recommend the above order)

Defining a class

- An object's data is stored in fields (also called instance variables) The fields describe the state of the object
- Fields are defined with ordinary variable declarations:

```
String name;
Double health;
int age = 0;
```

- Instance variables are available *throughout the entire class* that declares them

Defining constructors

- A constructor is code to create an object
  Please note: you can do other work in a constructor, but you shouldn't
- The syntax for a constructor is:

```
ClassName(parameters) {
    code
}
```

- The `ClassName` has to be the same as the class that the constructor occurs in
- The `parameters` are a comma-separated list of variable declarations

Example constructor I

```
package person1;

public class Person {
  String name;
  int age;
  boolean male;

  Person(String aName, boolean isMale) {
    name = aName;
    male = isMale;
  }
}
```

Example constructor II

```
package person2;

public class Person {
  String name;
  boolean male;

  Person(String name, boolean male) {
    this.name = name;
    this.male = male;
  }
}
```

Defining a method

A method has the syntax:

```
return-type  method-name(parameters) {
    method-variables
    code
}
```

Example

```
boolean isAdult(int age) {
    int magicAge = 21;
    return age >= magicAge;
}
```

Example

```
double average(int a, int b) {
    return (a + b) / 2.0;
}
```

Methods may have local variables

- A method may have local (method) variables
- Formal parameters are a kind of local variable

```
int add(int m, int n) {
    int sum = m + n;
    return sum;
}
```

- `m`, `n`, and `sum` are all local variables
  - The scope of `m`, `n`, and `sum` is the method
  - These variables can only be used in the method, *nowhere else*
  - The *names* can be re-used elsewhere, for *other* variables

Scoping situations

- Block (or compound statements)
- Declarations in a method
- Nested scopes
- The `for` loop (as we've already discussed)

Returning a result from a method

If a method is to return a result, it must specify the type of the result:

```
boolean isAdult ( ...
```

You must use a `return` statement to exit the method with a result of the correct type:

```
return age >= magicAge;
```

Returning *no* result from a method

- The keyword `void` is used to indicate that a method does not return a value
- The `return` statement must not specify a value

Example

```
void printAge(String name, int age) {
    System.out.println(name + " is " + age + " years old.");
    return;
}
```

- There are two ways to `return` from a `void` method:
  1. Execute a `return` statement
  2. Reach the closing brace of the method

Sending messages to objects

- We do not perform operations on objects, we *talk* to them
- This is called sending a message to the object
- A message looks like this:
  `object.method(extra information)`
- The *object* is the thing we are talking to
- The method is a name of the action we want the object to take
- The extra information is anything required by the method to do its job

Example

```
g.setColor(Color.pink);
amountOfRed = Color.pink.getRed( );
```

The Person class revisited

```
package person3;

public class Person {
  // fields with access modifiers
  private String name;
  private int age;

  // constructor
  public Person(String name){
    this.name = name;
    age = 0;
  }

  // methods

  public String getName(){
    return name;
  }

  public void birthday(){
    age++;
    System.out.println("Happy birthday!");
  }
}
```

Using our new class

```
package person3;

public class Tester {
  public static void main(String[] args){
    Person john;
    john = new Person("John Smith");

    System.out.print(john.getName());
    System.out.println(" is having a birthday!");
    john.birthday();
  }
}
```

null

- If you declare a variable to have a given object type, for example,

```
Person john;
String name;
```

- . . . and if you have not yet assigned a value to it, for example, with

```
john = new Person();
String name = John Smith";
```

- . . . then the value of the variable is `null`
  - `null` is a legal value, but there is not much you can do with it
  - It is an error to refer to its fields, because it has none
  - It is an error to send a message to it, because it has no methods
  - The error you will see is `NullPointerException` (ring any bells?)

Methods and static methods

Java has two kinds of methods:

- static methods and
- non-static methods (called instance methods)

However, before we can talk about what it means to be static, we have to learn a lot more about classes and objects
Most methods you write should not, and will not be static
Every Java program that you wish to execute must have a method

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

This starts us in a *static context*

Classes and methods lead to *Modularization*

- Whenever a program is broken into two parts, there comes into being an *interface* between them:
  - The parameter list
  - Any global or common
  - More subtle ways of information transmission
- This interface should be kept as *narrow* and as *explicit* as possible.
- How to make it narrow?
  - Have each method do only one thing
  - Be able to describe what it does without saying *but* or *except*
  - Keep parameter lists short
  - Avoid global variables
- How to make it explicit?
  - Avoid global variables
  - Avoid side effects

Birkbeck UNIVERSITY OF LONDON

Classes lead to *Information Hiding*

- The principle of information hiding is, informally,

  Every method should mind its own business!

- To use a method, you need to know:
  - What information do you need to give it?
  - What does it tell you (or do for you) in return?
- You should not need to know:
  - How it does its job
- The method should not need to know:
  - How you do your job

Birkbeck UNIVERSITY OF LONDON

A fragment of code...

Consider the following method:

```
public class Play {
  // ...
  void play() {
    setup();
    player = who_goes_first();
    do {
      if (player == HUMAN) {
        do {
          move = get_humans_move();
          check_if_legal(move);
          movecounter++;
        } while (!ok);
        make_move(move);
      } else { /* player == COMPUTER */
        move = choose_computers_move();
        make_move(move);
      }
    } while (!game_over);
  }
}
```

How is this code unclear?

The *not-so-good* program

- What routine or routines update `player`?
- Do they do it in such a way that the main program works?
- If the human's move is not `ok`, has `player` already been updated?
- How does `make_move()` know whose move to make?
- Who sets `ok`?
- When does the game end?
- Who is responsible for deciding this?
- Is `movecounter` being
  - Initialised properly?
  - Computed properly?
  - What is it anyway, and who uses it?

Birkbeck UNIVERSITY OF LONDON

Class variables and methods I

- A class describes the variables and methods belonging to objects of that class
- These are called instance variables and instance methods
- A class may also have *its own* variables and methods
- These are called class variables and class methods
- initialisation blocks — static and instance

Birkbeck UNIVERSITY OF LONDON

Class variables and methods II

Why have class variables and methods?

- Sometimes you want to keep information about the class itself
- Example: Class `Person` might have a class variable `population` that keeps a count of the number of people
- This would not be appropriate data to keep in each `Person`!
- Sometimes you want to do something relevant to the class as a whole
- For example, find the *average age of a population*
- Sometimes you do not have any objects
- For example, you want to start up a program

Birkbeck
UNIVERSITY OF LONDON

Example use of a class variable

```
package person4;

public class Person {
  // fields
  private String name;
  private int age;
  private static int population;

  // static initialization block

  static {
    population = 0;
  }

  // initialization block

  {
    age = 0;
  }

  // constructor
  public Person(String name){
    this.name = name;
    population++;
  }

  // methods

  public String getName(){
    return name;
  }

  public void birthday(){
    age++;
    System.out.println("Happy birthday!");
  }

  public static int getPopulation(){
    return population;
  }
}
```

Questions