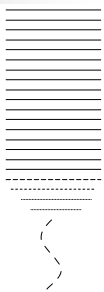


# 1B1a

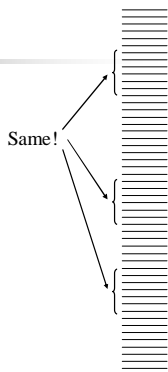
## Methods


1 Copyright © 2003, Graham Roberts Department of Computer Science

- ### Review so far
- We have:
    - sequence, iteration, selection.
    - variables, assignment, operators.
    - expressions, statements and compound statements.
    - and other useful bits.
  - We want to build classes.
  - But we need more building blocks...
- 2 Copyright © 2003, Graham Roberts Department of Computer Science

- ### Longer programs
- So far we have written rather small programs.
  - We want to write bigger and more interesting programs!!
  - Really long programs.
  - But using *only* loops and selection doesn't scale up.
- 
- 3 Copyright © 2003, Graham Roberts Department of Computer Science

- ### But...
- We start to get management problems.
  - Working with a huge long list of simple statements is tedious.
  - Worse, it is hard to see what the program does without plodding through the detail.
  - There's just no real *structure*...
- 4 Copyright © 2003, Graham Roberts Department of Computer Science

- ### Repetition...
- The same sub-sequence of statements often occurs in several places...
  - And loops don't help.
  - Bad news - we don't want to have duplicate lines of code.
- 
- 5 Copyright © 2003, Graham Roberts Department of Computer Science

- ### Suppose...
- We take a sub-sequence of statements and give it a name.
- square:  } Print a square of stars
- And package the name and statements together.
- 6 Copyright © 2003, Graham Roberts Department of Computer Science

## Abstraction in action!

- We can now refer to 'square' to denote the statement sequence.
- Write the sequence once and refer to it from many places in the program.
- Wow - we've exploited abstraction!

7

Copyright © 2003, Graham Roberts

Department of Computer Science



## Method

- The name + statement sequence gives us a *method* (a routine to do something).
- We can use the name to *call* the method on an object:

```
obj.square();
```

Note the parentheses. We use them to denote a call being made.

8

Copyright © 2003, Graham Roberts

Department of Computer Science



## Calling you...

```
obj.square();
obj.square();
obj.square();
```

- The method can be repeatedly called as many times as we want.
- But only needs to be written and debugged *once*.

9

Copyright © 2003, Graham Roberts

Department of Computer Science



## Procedures, functions?

- Some programming languages have procedures or functions don't they?
- Yes, but Java is Object-Oriented.
- Objects have methods.
- A method is called for an object.
  - Remember robot.forward() etc.

10

Copyright © 2003, Graham Roberts

Department of Computer Science



## Methods and Classes

```
class MyClass
{
    public void methodOne()
    {
        // Statements
    }
    public void methodTwo()
    {
        // Statements
    }
    etc...
}
```

A class can declare a number of methods. Each method can be called on an *instance* object of the class.

```
MyClass obj = new MyClass();
obj.methodOne();
obj.methodTwo();
```

11

Copyright © 2003, Graham Roberts

Department of Computer Science



## So let's write a square method...

```
class MyClass
{
    public void square()
    {
        for (int i = 0 ; i < 4 ; i++)
        {
            System.out.println("*****");
        }
    }
    // More methods can be added here.
}
```

Note that the method is declared in a (very simple) class.

Hey, I've been writing methods all the time!

12

Copyright © 2003, Graham Roberts

Department of Computer Science



## public void

- *public* - the method can be called from *anywhere* within a program. Providing it is called on an object that is accessible.
- *void* - is a type, the empty type that has no values. It means our method returns no value.

13 Copyright © 2003, Graham Roberts

Department of Computer Science



## Calling square()

```

Class MyClass
{
    // ...
    public static void main(String[] args)
    {
        MyClass myObject = new MyClass();
        myObject.square();
        myObject.square();
        myObject.square();
    }
}

```

Create an object using `new`, then call the square method.

Remember, a class defines how an instance object is implemented.

14 Copyright © 2003, Graham Roberts

Department of Computer Science



## main?

- The method named main has a special role.
  - It is public.
  - Is void (returns no value).
  - It is the first method called to start the program running.
    - Create an object, call methods.
  - And *static*...

15 Copyright © 2003, Graham Roberts

Department of Computer Science



## Static?

- A static method does *not* need to be called for an object.
- Main has to be static as there are no objects available before it is called...
- Main creates the initial object(s) needed by the program and calls their methods.
  - The object(s) then do the work.

16 Copyright © 2003, Graham Roberts

Department of Computer Science



## Other static methods?

- *Your* exercise classes only ever need to declare one static method – main!
- Don't write any other static methods.
- (Static methods do have valid uses as providers of utility functions and services.)

17 Copyright © 2003, Graham Roberts

Department of Computer Science



## What is a program?

- Our view of what a program is starting to develop:
  - Not simply a sequence of statements.
  - But a collection of objects described by classes, where the objects call each other's methods.

18 Copyright © 2003, Graham Roberts

Department of Computer Science



## Writing a program

- Identify classes needed.
- Identify methods and variables needed by objects of those classes.
- Write classes.
- Run program by creating object(s) in a main method, and then calling object methods.
  - Objects then call each other's methods.

19 Copyright © 2003, Graham Roberts

Department of Computer Science



## Multiple Main Methods

- Each class can have a main method.
  - But only one per class.
- Typically one is used to run the program.
  - The rest can be used for things like testing objects of each class.
  - Or showing examples of how to use the class.

20 Copyright © 2003, Graham Roberts

Department of Computer Science



## Questions?

21 Copyright © 2003, Graham Roberts

Department of Computer Science



## Methods calling methods

```
class MyClass
{
    public void square()
    {
        // As already seen
    }
    public threeSquares()
    {
        square();
        square();
        square();
    }
    etc...
```

square() ; – where's the object?

22 Copyright © 2003, Graham Roberts

Department of Computer Science



## Same class

- When a method calls another method in the *same* class:
  - It is called for the same object.
- So: square()
  - called for the same object threeSquares was called for.

23 Copyright © 2003, Graham Roberts

Department of Computer Science



## Main...

```
public static void main(String[] args)
{
    MyClass myObject = new MyClass();
    myObject.threeSquares();
}
```

Here is the object.

square will be called for the same object.

24 Copyright © 2003, Graham Roberts

Department of Computer Science



## this

Could write:

```
public threeSquares()
{
    this.square();
    this.square();
    this.square();
}
```

*this* is a variable referring to the object the method was called on.

this is *automatically* declared in every method.

25 Copyright © 2003, Graham Roberts

Department of Computer Science



## This or not to this...

- Sometimes you have to use `this`.
- Most of the time it is a matter of style.

26 Copyright © 2003, Graham Roberts

Department of Computer Science



## Very nice but...

- `square` can only display a 4x4 square.
- (And I cheat by displaying a whole line of stars at one go!)
- How can we modify `square` to print any size square?

27 Copyright © 2003, Graham Roberts

Department of Computer Science



## Parameters

- We want to *parameterise* `square`.
- That is, be able to write:
 

```
    this.square(3); // 3x3 square
    this.square(10); // 10x10 square
```
- We can say that `square` takes a *parameter* or *argument*.

28 Copyright © 2003, Graham Roberts

Department of Computer Science



## Square.2

```
public void square(int size)
{
    for (int rows = 0 ; rows < size ; rows++)
    {
        for (int cols = 0 ; cols < size ; cols++)
        {
            System.out.print("*");
        }
        System.out.print("\n");
    }
}
```

29 Copyright © 2003, Graham Roberts

Department of Computer Science



## Parameter variable

*size* is a *parameter variable*

```
public void square(int size)
```

- When `square` is called – `this.square(5)` – the argument is used to *initialise* the parameter variable.
- Inside the *method body*, the value of *size* can be used (or changed by assignment).
- *size* is created and initialised by every call to `square`.

30 Copyright © 2003, Graham Roberts

Department of Computer Science



## Parameterisation

- We now have a way of varying the behaviour of a method depending on which argument value it is called with.
- Square is now more general purpose.
- A better abstraction.

31 Copyright © 2003, Graham Roberts

Department of Computer Science



## Where can size be used?

```
public void square(int size)
{
    // size useable here only
}
```

- The parameter variable can *only* be used inside the method.
- In fact, it only exists while the method is executing.

32 Copyright © 2003, Graham Roberts

Department of Computer Science



## Scope revisited

- We say that the *scope* of `size` is given by the method body (which is a compound statement).
- A name can only be used inside the scope it is declared in.
- Nothing outside the scope can see the name or use it.

33 Copyright © 2003, Graham Roberts

Department of Computer Science



## Lifetime revisited

```
public void square(int size)
{ ... }
```

- When the scope is *entered*, `size` is created and initialised.
- When the scope is *exited*, `size` is thrown away.
- The scope determines the *lifetime* of `size`.

34 Copyright © 2003, Graham Roberts

Department of Computer Science



## New size every time

```
public void square(int size)
{ ... }
```

- Whenever the method is called, a new scope is created.
- *And* a new `size` variable.
- Once the scope is exited the current `size` variable is gone for good.

35 Copyright © 2003, Graham Roberts

Department of Computer Science



## Compound statement (reminder)

- When you see a compound statement:
 

```
{
    // Statements ...
}
```

you are seeing a scope.

- This includes loop bodies, if statement bodies and class declarations.

36 Copyright © 2003, Graham Roberts

Department of Computer Science



## Local scope (reminder)

```
while (n < 10)
{
  int x = n + 1 ;
  String s = "Hello";
  ...
}
```

A local scope

A Local Variable

37 Copyright © 2003, Graham Roberts

Department of Computer Science



## Local variables - temporary storage

- Declared within a compound statement, including a method body.
- Local to that scope.
- Lifetime limited to that scope.
- Created when scope is entered.
- Thrown away when scope is exited.
- Every time!

38 Copyright © 2003, Graham Roberts

Department of Computer Science



## Scope must be obeyed

```
// result not declared before loop
while (x < y)
{
  int result = 0;
  ... // Do something
}
result++; // ERROR, result not in scope
```

39 Copyright © 2003, Graham Roberts

Department of Computer Science



## Nesting Scopes

- You can *nest* compound statements, so scopes are *nested* as well.
- ```
{
  // Outer or nesting scope
  {
    // Inner or nested scope
  }
}
```

40 Copyright © 2003, Graham Roberts

Department of Computer Science



## Nested Scopes (2)

```
{
  String day = "Monday";
  ...
  {
    day = "Tuesday"; // day is accessible in
  } // nested scopes
  ...
}
```

41 Copyright © 2003, Graham Roberts

Department of Computer Science



## But...

```
{
  String day = "Monday";
  ...
  {
    String day = "Tuesday"; // ERROR day is
    ... // already in scope
  }
  ...
}
```

42 Copyright © 2003, Graham Roberts

Department of Computer Science



## For loop scope rule

```
for (int i = 0 ; i < 10 ; i++) // i is a for loop variable
{
    System.out.print(i + " ");
}
i = 6; // Error, not in scope
for (int i = 0 ; i < 10 ; i++) // i can be re-declared here.
{
    // The two i's are different
    System.out.print(i + " "); // and the scopes disjoint.
}
```

43 Copyright © 2003, Graham Roberts

Department of Computer Science



## Names can be reused

- Providing the same name is declared in disjoint scopes, the name can be reused.
- If you are unsure about how the rules apply, write some test programs to try them out.

44 Copyright © 2003, Graham Roberts

Department of Computer Science



## Questions?

45 Copyright © 2003, Graham Roberts

Department of Computer Science



## More parameters

- A method can have any number of parameters.
- But good practice means no more than 5 or 6 maximum.

46 Copyright © 2003, Graham Roberts

Department of Computer Science



## Another method

```
public void rectangle(int nRows, int nCols, char c)
{
    for (int rows = 0 ; rows < nRows ; rows++)
    {
        for (int cols = 0 ; cols < nCols ; cols++)
        {
            System.out.print(c);
        }
        System.out.print("\n");
    }
}
```

```
rectangle(3,7,'#') ;
rectangle(10,4,'+') ;
```

47 Copyright © 2003, Graham Roberts

Department of Computer Science



## Parameter Types

- ```
rectangle(int nRows, int nCols, char c)
```
- Parameter variables are declared with types.
  - The values supplied in the method call must have matching types:
 

```
rectangle(5,7,'c'); // OK
rectangle(2.3,5,"hello"); // Error!!
```

48 Copyright © 2003, Graham Roberts

Department of Computer Science



## Typical error message

compiling: Rectangle.java

Rectangle.java:7: Incompatible type for method. Explicit cast needed to convert double to int.

```
rectangle(4.5,6,'#');
           ^
```

1 error

49 Copyright © 2003, Graham Roberts

Department of Computer Science



## Remember main?

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

- main is called to run your program.
- It takes a String array as an argument.
- What is the contents of the array when the method is called?

50 Copyright © 2003, Graham Roberts

Department of Computer Science



## Command line arguments

- When you run a program you can type additional arguments on the command line:

```
java MyProg hello world
```

Arguments to the program.

51 Copyright © 2003, Graham Roberts

Department of Computer Science



## Displaying the arguments

```
public static void main(String[] args)
{
    for (int n = 0 ; n < args.length ; n++)
    {
        System.out.println(args[n]);
    }
}
```

52 Copyright © 2003, Graham Roberts

Department of Computer Science



## You see:

```
Prompt> java Args a few words
a
few
words
Prompt>
```

Arguments displayed here.

53 Copyright © 2003, Graham Roberts

Department of Computer Science



## Functions

- Suppose we want a method that looks like a function.
  - Like Math.cos(x), for example.
- Need to be able to do a calculation and *return* a value from a method.

54 Copyright © 2003, Graham Roberts

Department of Computer Science



## return

- The return statement allows a method to return a value:

```
public int f(int x)
{
    x *= 2;
    return x;
}
```

Return the value of x.

Declare the type of the value returned

55 Copyright © 2003, Graham Roberts Department of Computer Science

## Effect of return

- Can now write: `int y = myObject.f(10);`
- Return causes the method to terminate, and return a value.
- (Effectively a jump to the end of the method body.)
- The type of the returned value must be declared.

56 Copyright © 2003, Graham Roberts Department of Computer Science

## Must return a value

- If a method is declared as returning a type, then it must contain a return statement.
- Otherwise the compiler will complain!

T2.java:8: Return required at end of int f(int).

```
int f(int x)
^
1 error
```

57 Copyright © 2003, Graham Roberts Department of Computer Science

## Can have multiple returns

```
public int f(int x)
{
    if (x < 10)
    {
        return x * 2;
    }
    else
    {
        return x * 3;
    }
}
```

Notice that regardless of what the value of x is, one return statement will *always* be executed.

This must always be the case.

58 Copyright © 2003, Graham Roberts Department of Computer Science

## Also

```
public int f(int x)
{
    return x * 2;
    System.out.println(x);
}
```

The print statement can never be executed!

The compiler will treat this as an error. It uses *flow analysis* to determine what can be executed.

59 Copyright © 2003, Graham Roberts Department of Computer Science

## Only one value

- A method can only return *one* value.
- Methods declared *void* don't return values.
  - `public void f(int x) { ... } // No return`
- Void is the empty type and has no values that could be returned.
  - But can use return on its own to return from (terminate) a void method.

60 Copyright © 2003, Graham Roberts Department of Computer Science

## Methods as functions

- Methods can be written to look like mathematical functions:
  - sqrt, pow, sin, cos, log, etc.
- However, beware, not all function methods behave like mathematical functions...

61 Copyright © 2003, Graham Roberts

Department of Computer Science



## Mathematical functions

- Always return the *same* value when applied to the same argument(s).
  - Referential transparency.
- But methods can be written to return different values when called with the same argument(s).
- Methods can also have side-effects (e.g., doing input or output).

62 Copyright © 2003, Graham Roberts

Department of Computer Science



## Summary

- We now have methods.
- To make methods more useful we need parameters.
- Local variables, scope and lifetime, combine with compound statements and method bodies.
- Methods can return values.

63 Copyright © 2003, Graham Roberts

Department of Computer Science

