CITS1001 week 2
Class definitions

Arran Stewart

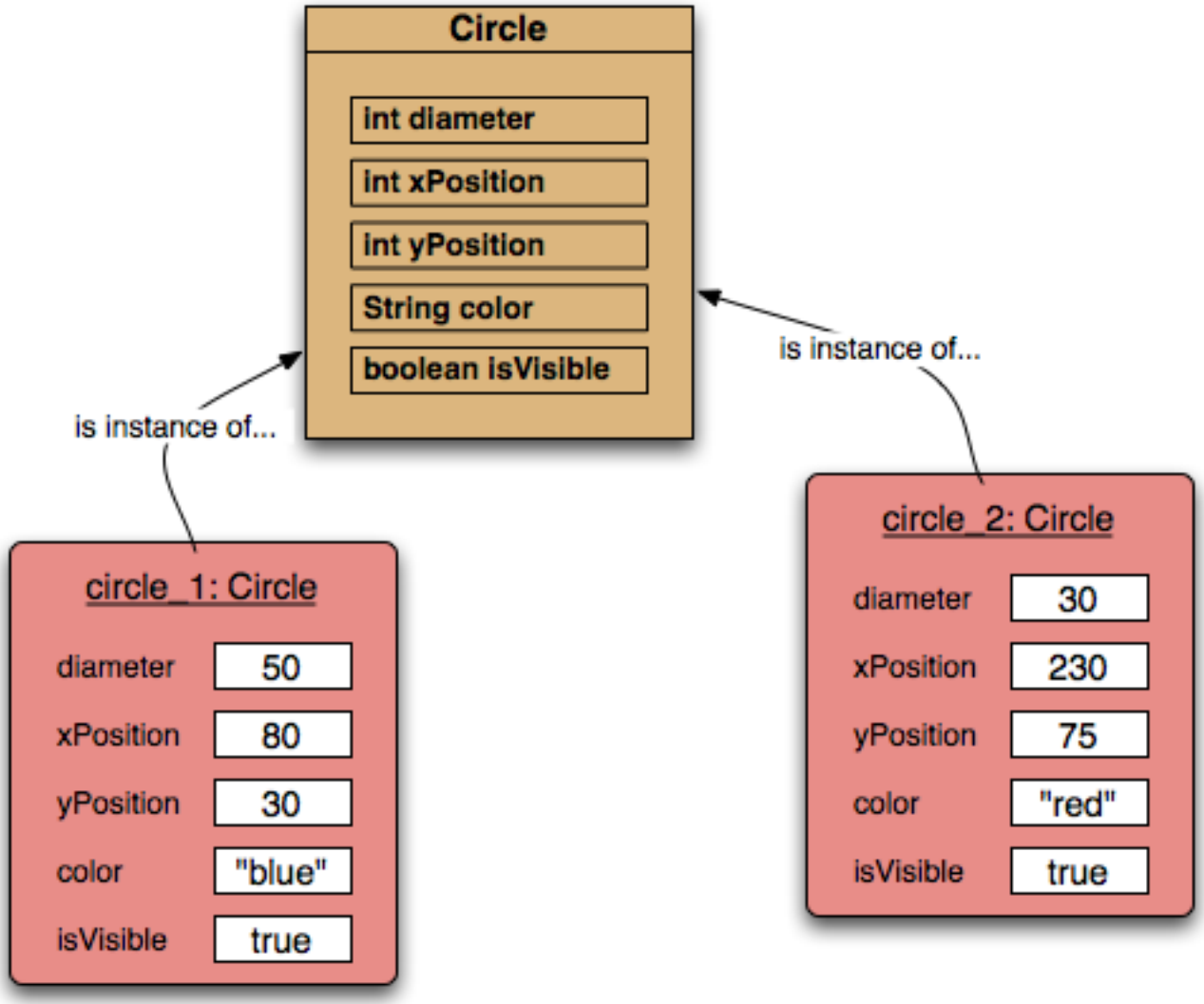
March 6, 2018

Week 1 revision

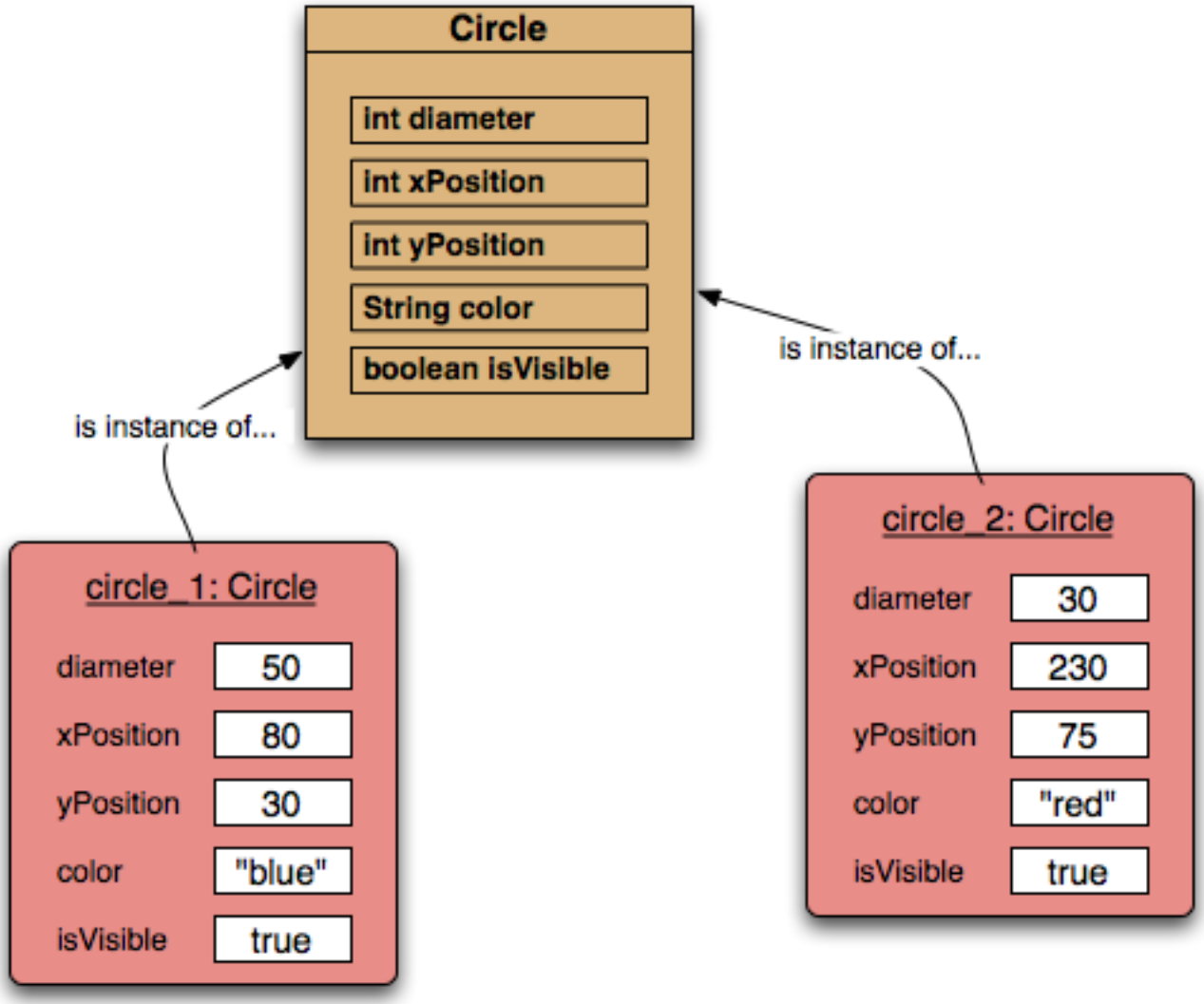
Revision of concepts from week 1

After studying the lectures, lab and reading Chapter 1, you should be familiar with the concepts of class, object, state and method

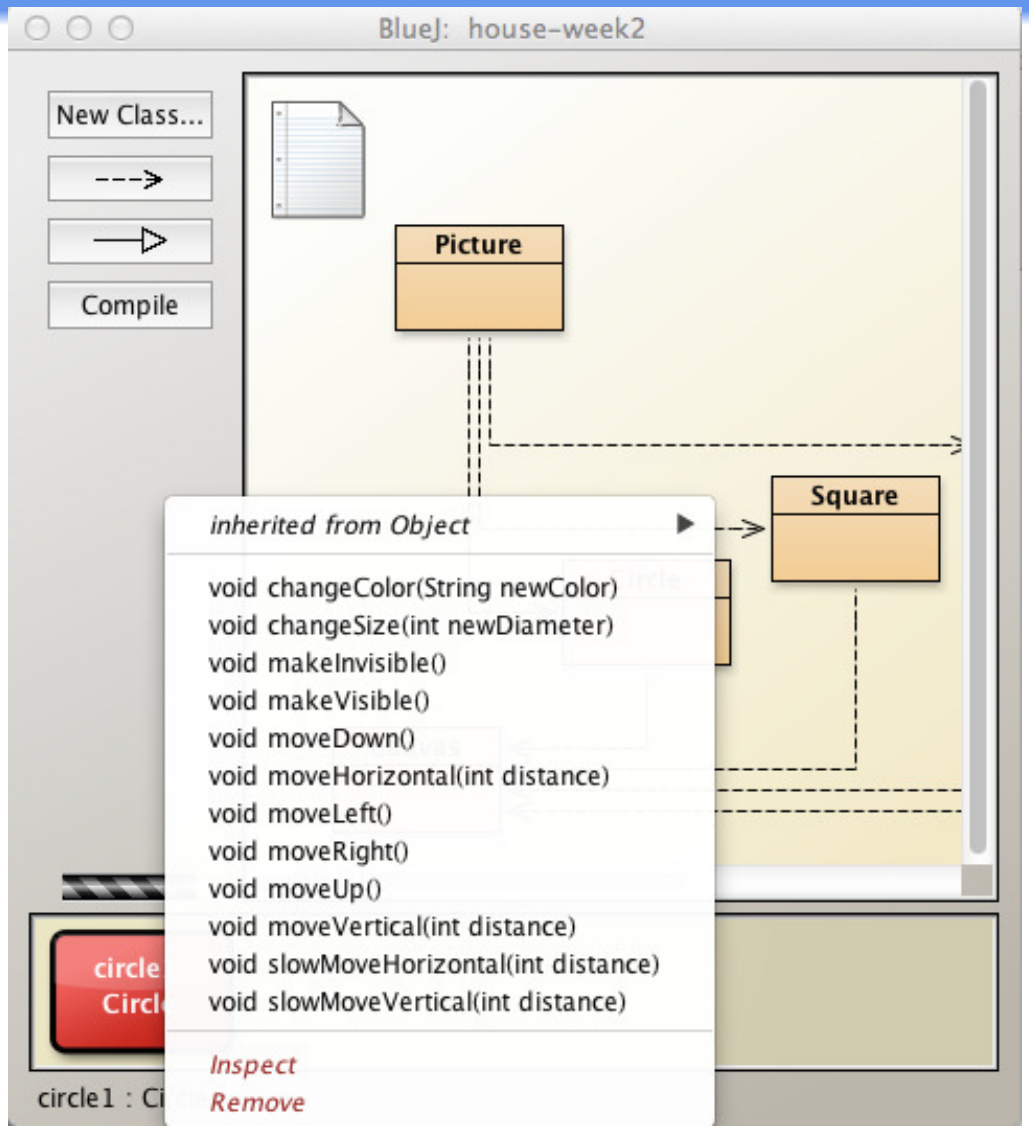
1. Objects are created by classes



2. Object state is represented by fields



3. Objects (usually) do something when we invoke a method



Methods can be thought of as *requests* we make of an object

Week 2 - Looking inside classes

Concepts (1)

This week we will learn to understand class definitions by looking inside Java classes.

- fields
- constructors
- comments

Reading: Chapter 2 of Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

This week we will learn to understand class definitions by looking inside Java classes.

- fields
- constructors
- comments

Reading: Chapter 2 of Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

- okay - first, a recap of classes

Classes vs. objects

- A **class** is a group of *objects* that have similar characteristics and that exhibit similar behaviour
- An **object** is a specific *instance of* a *class*

- Classes represent all objects of a certain kind
  - e.g. Car, Lecturer, Student
- Objects represent 'things' from the real world, or from some problem domain
  - e.g. the red car down there in the car park
  - e.g. the lecturer talking to you now

Classes vs. objects (2)

- Classes are like a “blueprint” or design for a set of objects:

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.
  - This blueprint or design exists even before any objects have been made.

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.
  - This blueprint or design exists even before any objects have been made.
- Objects are said to exist at "run-time":

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.
  - This blueprint or design exists even before any objects have been made.
- Objects are said to exist at "run-time":
  - When we start up a program, no objects exist

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.
  - This blueprint or design exists even before any objects have been made.
- Objects are said to exist at "run-time":
  - When we start up a program, no objects exist
  - The program creates objects as it runs, and methods of those objects are invoked to enact the behaviour of the program

Classes vs. objects (2)

- Classes are like a "blueprint" or design for a set of objects:
  - The source code we write in Java describes what sort of state and behaviour the objects of a class will have.
  - This blueprint or design exists even before any objects have been made.
- Objects are said to exist at "run-time":
  - When we start up a program, no objects exist
  - The program creates objects as it runs, and methods of those objects are invoked to enact the behaviour of the program
- This relationship is illustrated in this week's lab sheet

Why do we use classes?

- To reduce complexity
  - If we have many objects, all of which are constructed in a similar way, is it better to have one design that describes them all, or for each to be tailor-made?
- Often, we know how to deal with an object based purely on knowing its *class*, without knowing anything specifically about that particular *instance*
- For example, if we encounter a dog – i.e. an instance of the class Dog – we already have a basic understanding of how to deal with it, even if we have never previously met that particular dog
  - We know that it might bark, or bite, or wag its tail, based purely on knowing that it is a Dog
  - Barking, biting, and tail-wagging are best viewed as features of the class Dog, not of any individual dog

Source code

- In Java, classes are defined by text files of **source code**
- Source code is designed to be both
  - human readable, and
  - machine readable
- Source code must specify *every* detail about how objects belonging to a class behave
  - Computers are very fast but also very literal
- In this lecture we will use as a running example the following code listing on the CITS1001 site:
  - TicketMachine.java

An important point about program code

- Program code is designed to be human readable
  - Familiar words are used for programming constructs (`if`, `else`, `while`, `repeat`, `for`)
  - Indented format is similar to paragraphs and sections in text
  - Meaningful variable names suggest what they are intended to represent (e.g. price, mark, studentName)
- **and** program code is also executed by a computer
  - The computer will do exactly what it is **told** to do
  - The **rules** of the language determine **exactly** what happens when the program is run

The computer does not know what you intended the program to do

An important point about program code

- Program code is designed to be human readable
  - Familiar words are used for programming constructs (`if`, `else`, `while`, `repeat`, `for`)
  - Indented format is similar to paragraphs and sections in text
  - Meaningful variable names suggest what they are intended to represent (e.g. price, mark, studentName)
- **and** program code is also executed by a computer
  - The computer will do exactly what it is **told** to do
  - The **rules** of the language determine **exactly** what happens when the program is run

The computer does not know what you intended the program to do

- also worth stressing that you can, for instance, have a method called "moveLeft", and decide to write code that actually moves a circle to the right.
- You can have a class called Circle, which actually makes shapes that are triangles.
- This would be impolite to readers of your code; but the computer wouldn't care.

What is a programming language?

- A program for a computer to follow must be expressed completely unambiguously
- There are many different *programming languages* in which programs can be written
- In order to write a working program, you need to learn
  - the *vocabulary* and *syntax* of the language, so you can write statements that make sense
  - how to make *sequences* of legal statements that do simple tasks
  - how to express what you want the computer to do in a simple enough way to translate into the programming language
- Similar to learning the *words*, which form *sentences*, and allow you to write a *story*, when learning a human language

What is a programming language?

What is a programming language?

- A program for a computer to follow must be expressed completely unambiguously
- There are many different *programming languages* in which programs can be written
- In order to write a working program, you need to learn
  - the *vocabulary* and *syntax* of the language, so you can write statements that make sense
  - how to make *sequences* of legal statements that do simple tasks
  - how to express what you want the computer to do in a simple enough way to translate into the programming language
- Similar to learning the *words*, which form *sentences*, and allow you to write a *story*, when learning a human language

next slide - note that *bugs are not magic*

Bugs have logical reasons

- Programming can be difficult at first.
  Bugs can seem to come from nowhere, for no reason.
- But there is *always a logical reason behind a bug*.

Bugs have logical reasons

- Programming can be difficult at first.
  Bugs can seem to come from nowhere, for no reason.
- But there is *always a logical reason behind a bug*.

- next slide - APIs

Ticket machines – an external view

- An *external* view of a class means considering
  - What objects of the class do
  - How we create and use those objects
- For example, ticket machines accept money, and supply tickets at a fixed price
- Some questions about that behaviour:
  - How is that price determined?
  - How is 'money' entered into a machine?
  - How does a machine keep track of the money that has been entered?
- This is the view relevant to the *user* of a class
- Sometimes, you will be writing classes for use by other programmers (or yourself, at a later date);
  sometimes, you will be using classes other programmers have written.

Ticket machines – an external view

- An *external* view of a class means considering
  - What objects of the class do
  - How we create and use those objects
- For example, ticket machines accept money, and supply tickets at a fixed price
- Some questions about that behaviour:
  - How is that price determined?
  - How is 'money' entered into a machine?
  - How does a machine keep track of the money that has been entered?
- This is the view relevant to the *user* of a class
- Sometimes, you will be writing classes for use by other programmers (or yourself, at a later date);
  sometimes, you will be using classes other programmers have written.

- next slide - it's up to *you* to make it consistent
- examples:
  - suppose we have number of grammatical errors in a doc
  - or DOB and age

Ticket machines – an internal view

- An *internal* view of a class means considering
  - How it stores information
  - How it does things
- Looking inside allows us to determine how behaviour is provided or implemented
- This is the view relevant to the writer of a class

- All Java classes should have a consistent internal view

The four components of a class

- A class definition has four components
  - Its name – what is the class called?
  - Its fields – what information do we hold for each object, and how is it represented?
  - Its constructors – how are objects created?
  - Its methods – what can objects do, and how do they do it?

It is (usually) easiest to consider the four components in this order, whether you are writing your own class, or reading someone else's

The four components of a class

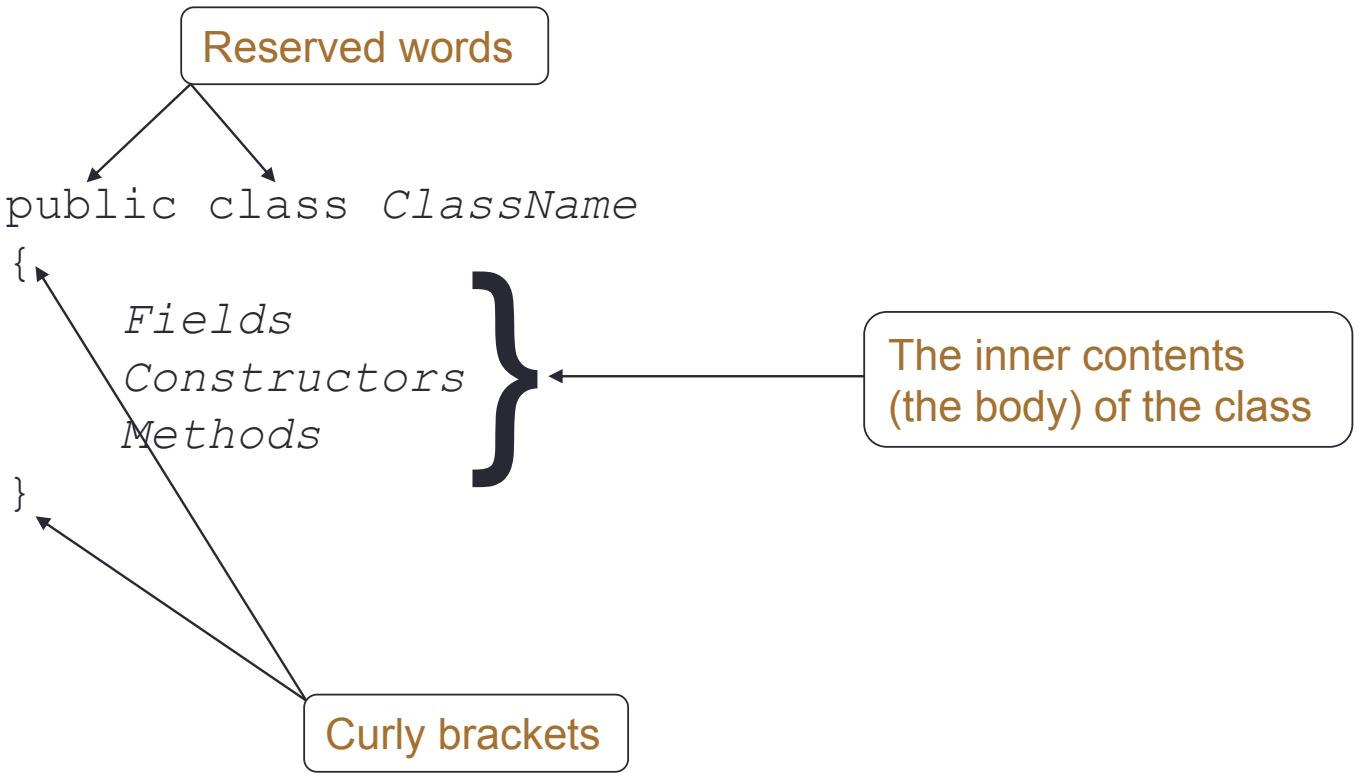
- next slide - this is a *typical* order
- some notes:
  - always be compiling
  - simplest class = empty

The four components of a class

- A class definition has four components
  - Its name – what is the class called?
  - Its fields – what information do we hold for each object, and how is it represented?
  - Its constructors – how are objects created?
  - Its methods – what can objects do, and how do they do it?

It is (usually) easiest to consider the four components in this order, whether you are writing your own class, or reading someone else's

Basic class structure



Syntax

- Reserved words and curly brackets are our first encounter with Java syntax
- Source code must be structured in a certain way, as determined by the rules of the language

- Reserved words are words with a special meaning in Java
  - e.g. public, class, private, int
  - There are many, many others
  - Also known as keywords
- Brackets (of all types) are everywhere in many languages
  - Here, they delimit the contents of the given class

- things that change rarely - ask for examples

Syntax

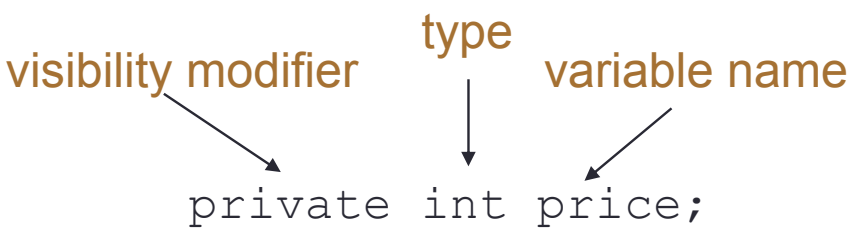
- Reserved words and curly brackets are our first encounter with Java syntax
- Source code must be structured in a certain way, as determined by the rules of the language

- Reserved words are words with a special meaning in Java
  - e.g. public, class, private, int
  - There are many, many others
  - Also known as keywords
- Brackets (of all types) are everywhere in many languages
  - Here, they delimit the contents of the given class

Fields

- Fields store values for an object.
- They are also known as instance variables.
- Fields define the state of an object.
- In BlueJ, we can use "Inspect" to view the state.
- Some values change often.
- Some change rarely (or not at all).

```
public class TicketMachine {
    private int price;
    private int balance;
    private int total;
    //...
}
```



The fields of TicketMachine

```
private int price;
private int balance;
private int total;
```

- Each field is described by a variable, which has
  - A visibility modifier, which denotes who can access it (more on this later)
  - A type, which denotes what values it can store (more on this later)
  - A name, chosen to make its use clear to human readers
- Additionally, and crucially, each field has a meaning
  - A sense of what information it stores
  - This should apply to every variable in every program you ever write
- Collectively, the fields denote the state of an object

Review questions

- What do you think is the *type* of each of the following fields?

```
private int count;
private Student representative;
private Server host;
```

- What are the *names* of the following fields?

```
private boolean alive;
private Person tutor;
private Game game;
```

Comments

- The other thing you will see in the source file TicketMachine.java is comments
- Comments are ignored by the computer; they exist simply to make the code easier for people to understand
- Comments come in three principal types
- Comments starting with //
  - In this case, the computer ignores everything up to the end of the line
- Comments starting with /*
  - In this case, the computer ignores everything up to the first occurrence of */, which acts like a closing bracket for the comment
- **Javadoc** comments start with /** and end with */
- We will discuss these later in the unit

Comments

- The other thing you will see in the source file TicketMachine.java is comments
- Comments are ignored by the computer; they exist simply to make the code easier for people to understand
- Comments come in three principal types
- Comments starting with `//`
  - In this case, the computer ignores everything up to the end of the line
- Comments starting with `/*`
  - In this case, the computer ignores everything up to the first occurrence of `*/`, which acts like a closing bracket for the comment
- **Javadoc** comments start with `/**` and end with `*/`
- We will discuss these later in the unit

- CONSTRUCTORS

Constructors

```
public TicketMachine(int cost) {
    price = cost;
    balance = 0;
    total = 0;
}
```

- Initialize an object.
- Have the same name as their class.
- Store initial values into the fields.
- Can use parameter values for this.

Constructors (2)

- The principal job of a constructor is to initialise the fields of the object
- Initial values may be
  - Set as defaults (e.g. balance, total), or
  - Derived from data passed in as parameters (e.g. price)
- Syntactically, the constructor is a special method
  - It has the same name as the class
  - It has no return type
- Note that there may be more than one constructor

Questions

- To what class does the following constructor belong?

```
public Student(String name)
```

- How many parameters does the following constructor have, and what are their types?

```
public Book(String title, double price)
```

- What do you think the types of the `Book` class's fields are? What about their names?

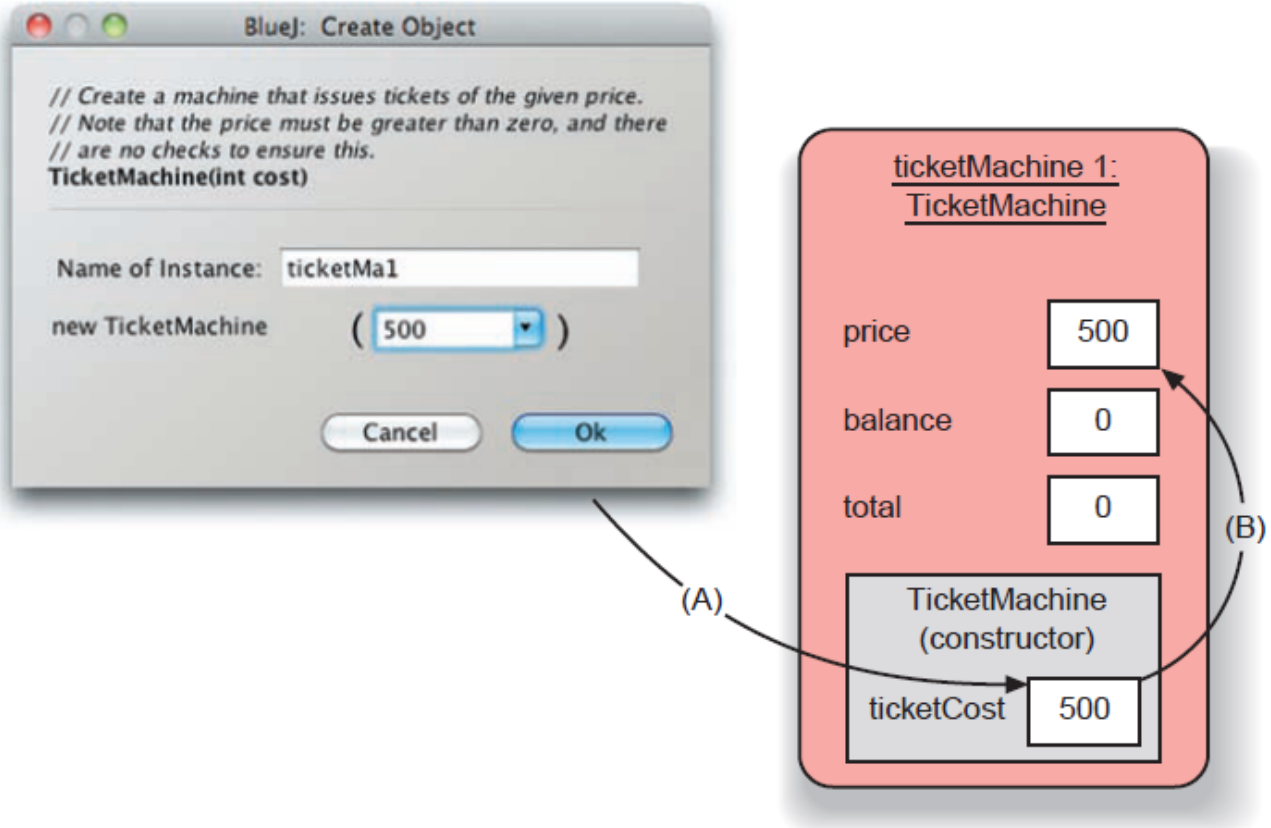
Aside: Default initialisation

- In Java, all fields are automatically initialised to a default value if they are not explicitly initialised.
- For integer fields, this default value is zero.
- However, we prefer to write the explicit assignments anyway.
- There is no disadvantage, and it serves to document what is actually happening.

Concepts for Constructors

- Parameters
- Scope of a variable
- Lifetime of a variable
- Assignment statements

Passing data via parameters



- Parameters are used by constructors and methods to receive values from outside.
- Parameters are another sort of variable

Parameters

- Parameter names inside a constructor or method are called formal parameters
- Parameter values outside are called actual parameters
- So cost is a formal parameter, and a user-supplied value such as 500 is an actual parameter
- Scope
  - The scope of a variable defines the section of source code from which the variable can be accessed.
- Lifetime
  - The lifetime of a variable describes how long the variable continues to exist before it is destroyed.

Scope and lifetime - parameters vs fields

- The scope of a formal parameter is restricted to the body of the constructor or method in which it is declared.
- The scope of a field is the whole of the class definition – it can be accessed from anywhere in the same class.
- The lifetime of a formal parameter is limited to a single call of the constructor or method.
- The lifetime of a field is the same as the lifetime of the object it belongs to.

Examples: see TicketMachine code

Choosing variable names

- There is a lot of freedom over choice of names. Use it wisely!
- Choose expressive names to make code easier to understand:
  - price, amount, name, age, etc.
- Avoid cryptic names:
  - w, t5, xyz123

Choosing variable names

- There is a lot of freedom over choice of names. Use it wisely!
- Choose expressive names to make code easier to understand:
  - price, amount, name, age, etc.
- Avoid cryptic names:
  - w, t5, xyz123

my rule: they're like nouns or pronouns. How complex the name should be depends on how far away you are from where it was defined. e.g. "The lecturer is here. Her car is there."

Assignment

- Values are stored into fields (and other variables) via assignment statements:

```
variable = expression;
price = cost;
```

- A variable stores a single value, so any previous value is lost.

exercise

- Suppose that the class `Pet` has a field called `name` that is of type `String`. Write an assignment statement in the body of the following constructor so the the `name` field will be initialized with the value of the constructor's parameter.

```
public Pet(String petsName) {
  // assignment statement goes here
}
```

Concepts for methods

- Methods
- Accessor methods
- Mutator methods

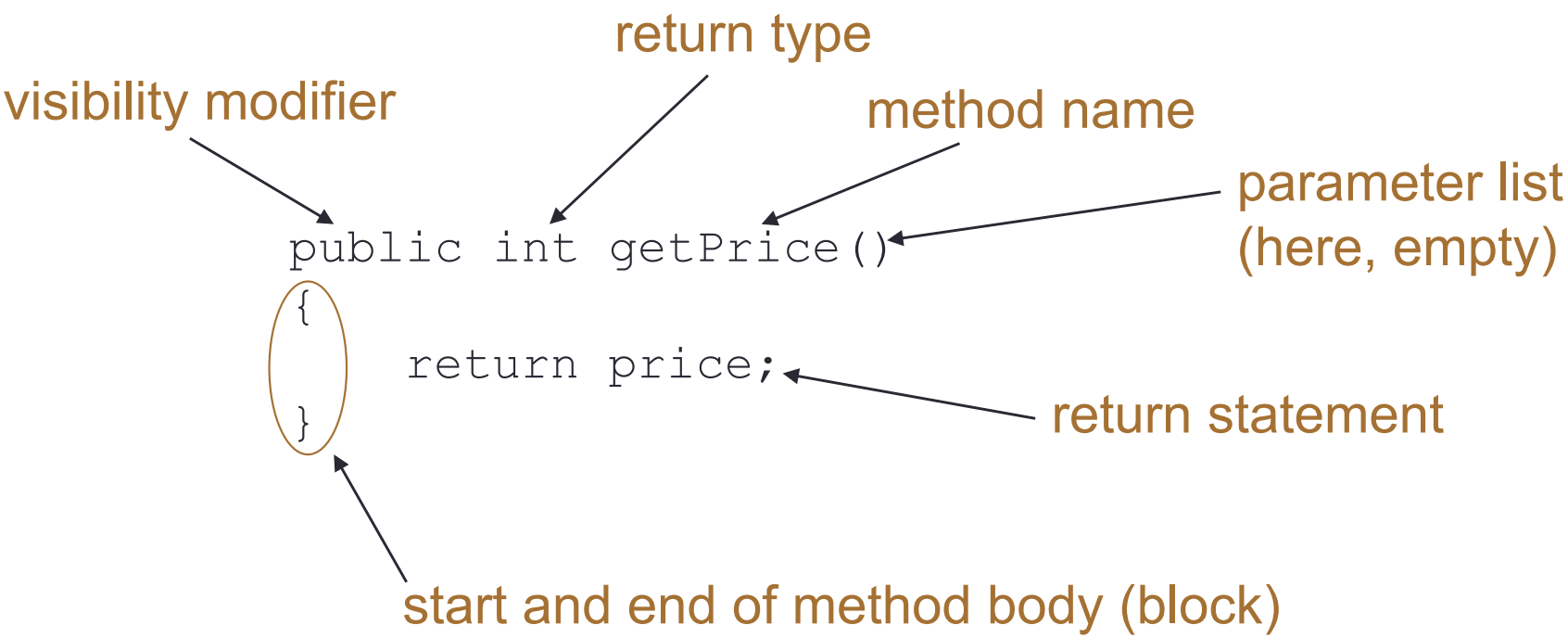
Methods

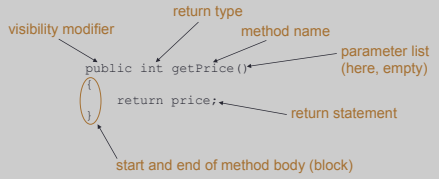
- Methods implement the behaviour of objects
- Methods have a consistent structure comprising
  - a header, and
  - a body
- Methods can implement any form of behaviour, as required by the class being implemented

Method structure

- The header provides the method's signature:
  - public int getPrice()
- The header tells us:
  - the name of the method
  - what parameters it takes
  - whether it returns a result
  - its visibility to objects of other classes
- The body encloses the method's statements.

Accessor (get) methods





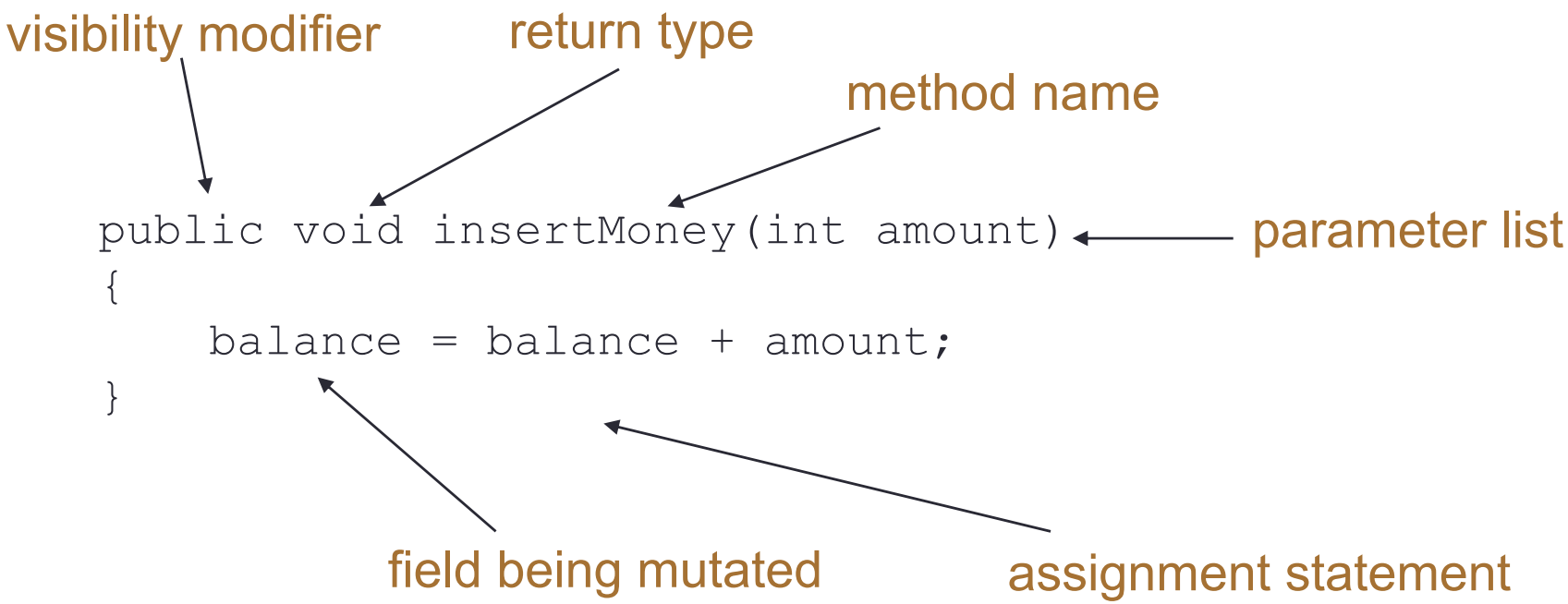
next slide:

- note commands vs queries
- “minor calculation” - e.g. might calculate age on the fly

Accessor methods

- An accessor method returns a value (result) of the type given in the header
- Usually it just looks up the current value of one of the object's fields
  - Sometimes it does some minor calculation on that value
- An accessor method always has a return type that is not void
- The method will contain a return statement to return the value
  - NB: returning is not printing!

Mutator methods



Mutator methods (2)

- They have the same method structure
  - Header and body
- They are used to mutate (i.e. change) an object's state
  - Achieved through changing the value of one or more fields
- They usually have the return type void
- They typically contain assignment statements
- They often receive data through parameters

set mutator methods

- Fields often have dedicated "set" mutator methods
- These have a simple, distinctive form
  - void return type
  - method name related to the field name
  - single parameter, with the same type as the type of the field
  - a single assignment statement

A typical set method

```
public void setDiscount(int amount) {
    discount = amount;
}
```

- We can infer from this that discount is probably a field of type int, i.e.

  ```
  private int discount;
  ```

Protective mutators

- A set method does not have to simply assign the parameter to the field
- The parameter may be checked for validity, and rejected if inappropriate
- Mutators thereby protect fields

Working with strings and output - printing

```
public void printTicket() {
    // Simulate the printing of a ticket.
    System.out.println("#####################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("#####################");
    total = total + balance;
    // Clear the balance.
    balance = 0;
}
```

String concatenation

We can concatenate strings using the same "+" operator used for numeric addition.

- 4 + 5
  9
- "wind" + "ow"
  "window"
- "Result:" + 6
  "Result: 6"
- "#" + price + " cents"
  "# 500 cents"

Quiz

If we try to concatenate a non-String value and a string, the value will be converted into a String (more on how this happens, later):

- System.out.println(5 + 6 + “hello”);
  11hello
- System.out.println(“hello” + 5 + 6);
  hello56

exercise

- How can we tell from just its header that `setPrice` is a method, and not a constructor?

  ```
  public void setPrice(int cost)
  ```

- Complete the body of the `setPrice` method so that it assigns the value of its parameter to the `price` field.

Method summary

- Methods implement all object behaviour
- A method has a name and a return-type
  - The return-type may be void
  - A non-void return type means the method returns a value to its caller
- A method might take parameters
  - Parameters bring values in from outside for the method to use
- Accessor methods provide information about an object
- Mutator methods alter the state of an object
- Other sorts of methods can accomplish a variety of tasks

Inside method bodies

Concepts

- conditional statements
- local variables

Reflecting on the ticket machines

- The behavior of the ticket machine objects so far is inadequate in several ways:
  - No checks on the amounts entered.
  - No refunds.
  - No checks for a sensible initialization.
- How can we do better?
  - We need more sophisticated behavior.

Making choices in everyday life

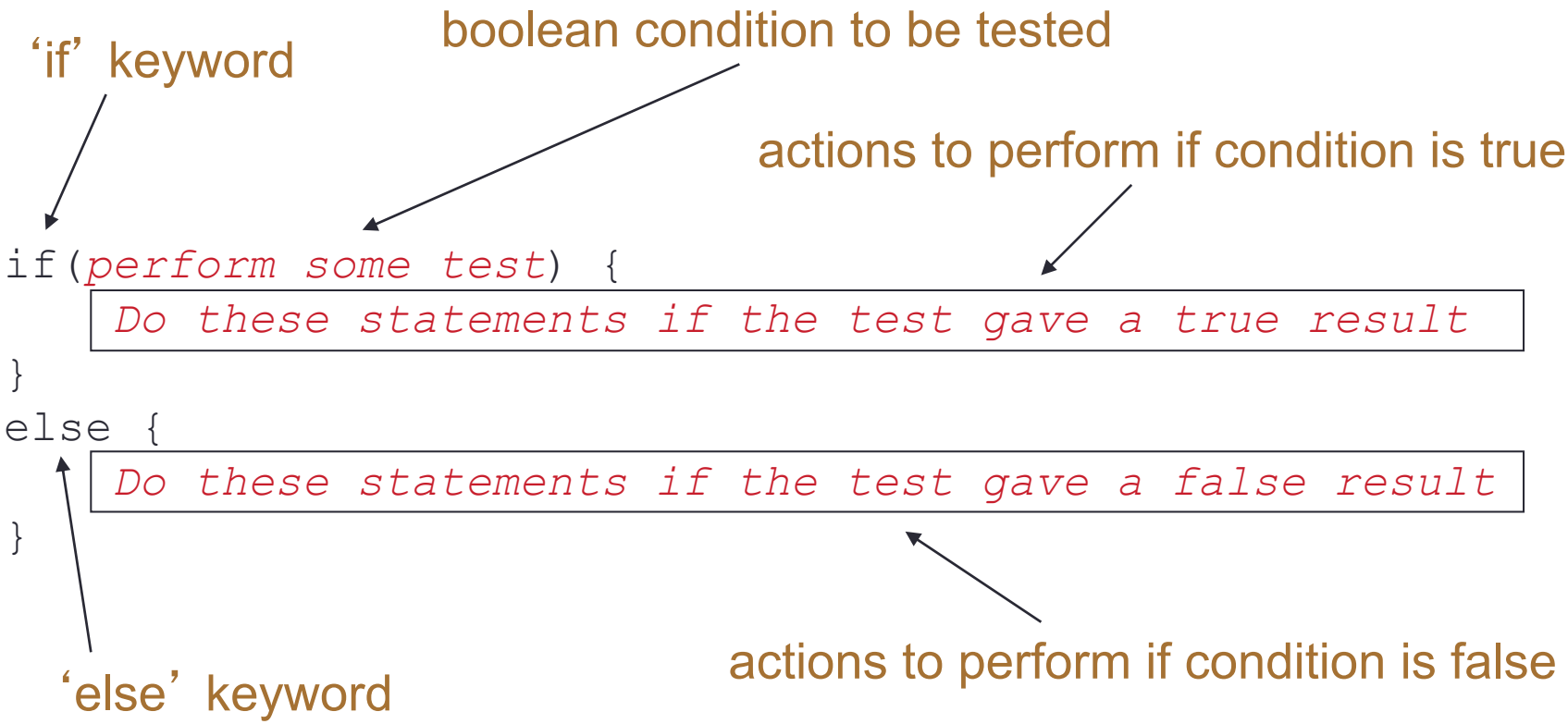
examples:

- "If I have enough money left, I will go out for a meal
  Otherwise, I will stay home and watch a movie"

```
if(I have enough money left) {
    // go out for a meal;
}
else {
    // stay home and watch a movie;
}
```

- The result depends on the amount of money available **at the time the decision is made**

Making choices in Java



Making a choice in the ticket machine

```
public void insertMoney(int amount) {
    if(amount > 0) {
        balance = balance + amount;
    }
    else {
        System.out.println(
            "Use a positive amount: " +
            amount);
    }
}
```

Print a ticket

- We'll examine the method for printing a ticket - `public void printTicket()`

Exercise

- Assume we have variables `price` and `budget`.
- Write an if statement that compares the value in `price` against the value in `budget`. If `price` is greater than `budget`, print the message "Too expensive"; otherwise print the message "Just right".

Variables – a recap

- Fields are one sort of variable.
  - They store values through the life of an object.
  - They are accessible throughout the class.
- Parameters are another sort of variable:
  - They receive values from outside the method.
  - They help a method complete its task.
  - Each call to the method receives a fresh set of values.
  - Parameter values are short lived.

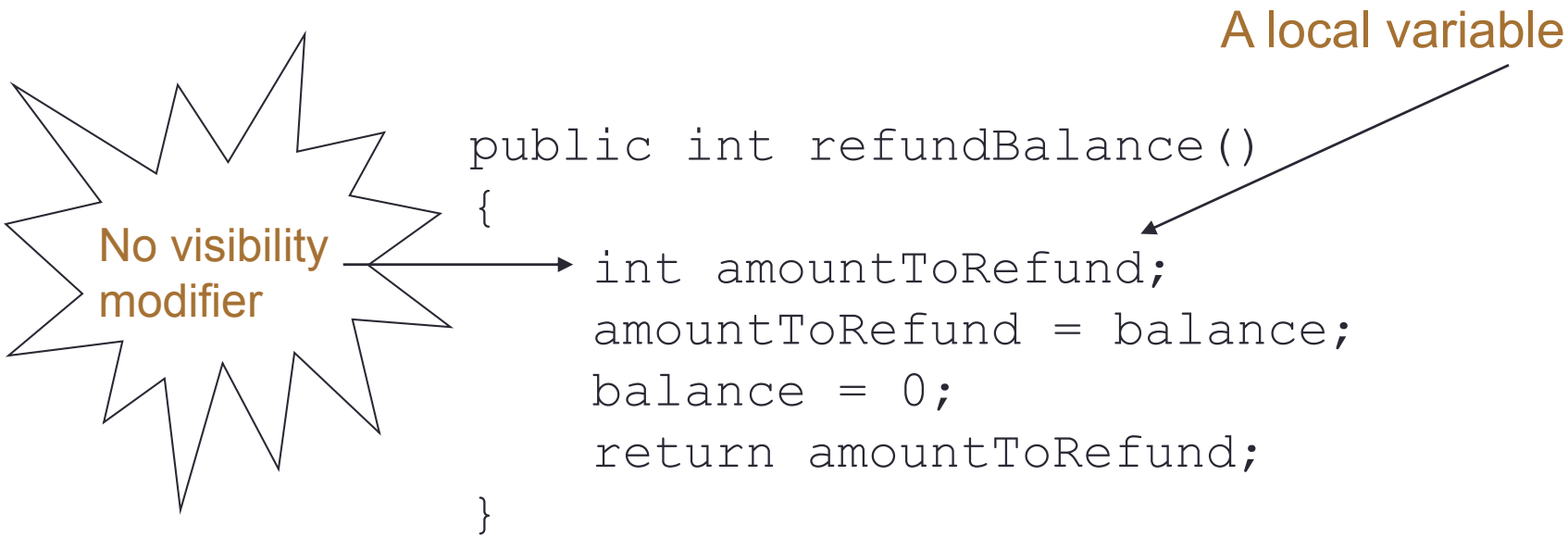
Local variables

- Methods can define their own, local variables:
  - Short lived, like parameters.
  - The method sets their values – unlike parameters, they do not receive external values.
  - Used for 'temporary' calculation and storage.
  - They exist only as long as the method is being executed.
  - They are only accessible from within the method.

Scope and lifetime

- Each block defines a new scope.
  - Class, method and statement.
- Scopes may be nested:
  - statement block inside another block inside a method body inside a class body.
- Scope is static (textual).
- Lifetime is dynamic (runtime).

Local variables [vs fields]



Local variables' scope and lifetime

- The scope of a local variable is the block in which it is declared.
- The lifetime of a local variable is the time of execution of the block in which it is declared.
- The scope of a field is its whole class.
- The lifetime of a field is the lifetime of its containing object.

How do we write 'refundBalance'?

- Return all the money left in the machine (balance) to the customer.
- And clear the balance to 0

refundBalance method

```
public int refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

The BlueJ debugger

The debugger

- Useful for gaining insights into program behavior . . .
- . . . whether or not there is a program error.
- Set breakpoints.
- Examine variables.
- Step through code.

Watch this introductory video:
Using the Debugger in BlueJ with Java
https://www.youtube.com/watch?v=w_iy0jmMmkA

Review (1)

- Class bodies contain fields, constructors, methods and comments.
- Field
  - Fields store data for an object to use. Fields are also known as instance variables.
- Constructors
  - Constructors allow each object to be set up properly when it is first created.
- Comment
  - Comments are inserted into source code to help human readers. They have no effect on the functionality of the code.

Review (2)

- Scope
  - The scope of a variable defines the section of source code from which the variable can be accessed.
- Lifetime
  - The lifetime of a variable describes how long the variable continues to exist before it is destroyed.
- Assignment
  - Assignment statements store the value represented by the right-hand side of the statement in the variable named on the left.

Review (3)

- Methods
  - Methods implement the behavior of objects.
- Accessor methods
  - Accessor methods return information about the state of an object.
- Mutator method
  - Mutator methods change the state of an object.
- println
  - The method `System.out.println()` prints its parameter to the text terminal.

Review (4)

- Conditional
  - A conditional statement takes one of two possible actions based upon the result of a test.
- Local variables
  - A local variable is a variable declared and used within a single method
  - Its scope and lifetime are limited to that of the method.
- Debugger
  - A debugger is a software tool that helps in examining how an application executes. It can be used to help find bugs.