

09s1COMP2911 Session 1 2009

Engineering Design in Computing

Week 1

John Potter

- Course Outline
- What is Design?
- Software Design
- OO Design
- Introduction to Java
- Java tools
- Java language
 - Java Virtual Machine
- Simple Examples
- Java Execution Model
- Objects on Heap
- Calls on Stack
- Execution Tracing in Debugger
- Static vs Non-static Methods
- Method Overloading
- Constructor Methods
- Access Modifiers: public vs private

Aims of Course

- basic principles of software design
 - abstraction
 - modularity
 - encapsulation and information hiding
- object oriented software development
 - concepts and terminology
 - basic object oriented design techniques
 - object oriented programming
 - implementing designs in Java
- design of algorithms
 - problem solving techniques
 - modelling and decomposing problems
 - implementing various strategies for finding solutions

Course Outline

- see Course Guide
 - general aims of course
 - schedule of lecture topics
- no set text
 - rely on lecture slides
 - online material
 - references will be provided
 - in lecture slides
 - and/or COMP2911 forum

Practical Work

- tutorials
 - discuss concepts and problems
 - getting good solutions is not the point of tutorials
 - focus on developing ideas and mental models
 - attendance + participation rewarded equally
- laboratories
 - developing practical experience
 - your tutor is your mentor
- lab assignments
 - test practical skills developed in labs
 - use concepts presented in lectures
- project
 - a software design and implementation exercise
 - preliminary design
 - revised design with interface specs
 - final implementation + unit testing
 - work in pairs within your lab groups

Assessment

- 5% Tutorial Attendance + Participation (10/12)
- 5% Lab Attendance + Participation (10/12)
- 30% 4 Assignments @ 7.5% each
 - due weeks 4, 6, 8, 12
- 30% Project in Groups of Two
 - 2.5% preliminary design
 - week 6
 - 5% revised design + interfaces
 - week 8
 - 22.5% final submission
 - Mon week 11
 - includes a demo of working application
 - week 11 or 12
- 30% Final Exam
- Total Course Mark
 - MAY BE capped at 45%
 - if 40% or less on final exam
 - if capped but average > 50%
 - passing a supplementary practical exam will remove the cap

What is Design?

What is Design?

- creative formulation of some artefact
- a description of something to be made or built

- two key elements

- **creativity**

- finding elegant solutions to problems
 - a search for beauty

- **technical description**

- designs are blueprints for construction
 - rely on engineering analysis to validate the design
 - feasibility - can it be built?
 - correctness - does it satisfy the requirements?



artistic side: *form*



scientific side: *function*

What is Good Design?

- which are examples of good design?
 - Sydney Opera House
 - Sydney Harbour Bridge
 - the computer mouse
 - the vinyl record
- characteristics of good design
 - beauty is subjective
 - functional effectiveness can be assessed
 - the best designs integrate a variety of creative elements with functional effectiveness
 - e.g. iPod, bicycle
- QWAN: "quality without a name"
 - Christopher Alexander: architectural design patterns
- often it's easier to recognise bad design
 - things just don't work like they should
 - we get annoyed when using them
 - when we try to repair things
 - when new designs highlight what we didn't know we were missing

Software Design

Software Requirements vs Software Design

- where do you start?
 - what's the problem?
 - what's the software needed for?
 - who is it needed for?
- requirements looks at the problem
- design outlines a solution to a problem

What's the problem?

- software is designed to provide a *solution* for some *problem*
 - software users will complain if the software does not address their needs
- *requirements analysis* tries to determine
 - system context
 - needs of users
- example
 - a lift control system
 - *users* include
 - passengers
 - maintenance personnel
 - *lift passengers* need
 - to request a lift from one floor to another
 - *maintenance people*
 - block user requests

Requirements Analysis

- focus on the problem
 - not the solution
- domain analysis
 - describe real-world context of the problem
- requirements elicitation
 - what do users want the software system to do?
 - should be described in terms of real-world interactions
- example
 - phone call forwarding
 - what does a user want?
 - delegation?
 - follow me?
 - when?
 - always?
 - on busy?
 - on no answer?

Software Design

- a design provides a solution to a problem
 - the problem statement should provide
 - real-world context
 - user needs
- a design specifies
 - *what* should be built
 - *not how* it is built
- example
 - design of a phone call forwarding system
 - specify possible states of the system
 - but keep it abstract
 - do not give exact details of the kind of data used to represent the state
 - and operations
 - accept user requests
 - accept and forward calls
 - how the system is changed by its users
 - how users get awareness of the state of the system

Requirements vs Design and Implementation

- Requirements
 - System context
 - Domain analysis
 - Where is the system to be used?
 - User needs
 - Requirements elicitation
 - What are interactions between system and user?
- Design
 - What does the system do?
 - Structure of components
 - Dependencies between components
- Implementation
 - How does the system work?
 - Details of data representation
 - Algorithmic details of behaviour

COMP2911 Focus: Design and Implementation

- Requirements
 - System context
 - Domain analysis
 - Where is the system to be used?
 - User needs
 - Requirements elicitation
 - What are interactions between system and user?
- Design
 - What does the system do?
 - Structure of components
 - Dependencies between components
- Implementation
 - How does the system work?
 - Details of data representation
 - Algorithmic details of behaviour

Typical Problems in Software Development

- finding out what is actually required
 - requirements engineering
- clear, concise and precise software design documentation
 - diagrammatic design notations
 - control flow (behaviour)
 - structured design (behaviour)
 - dataflow (DFDs) (behaviour)
 - entity relationship (data)
 - UML (data+behaviour)
 - formal specification techniques
 - precise mathematical / logical description of system structure and behaviour
- guaranteeing that software does what it should
 - verification
 - mathematical proof
 - for specialised applications and critical components
 - validation
 - testing cannot cover all possible cases
 - testing can prove the presence of bugs, not their absence

Typical Problems in Software Maintenance

- bug fixes
- new features
- changing requirements

- how do we minimise the impact of changes?
 - need to avoid the ripple effect
 - one change in one component propagates to changes throughout the system

- how do we know that changes preserve critical properties of an implementation?
 - programmers often code with implicit assumptions
 - these need to be made clear
 - design and code documentation has two key purposes
 - for users
 - for maintainers

OO Design

OO Design

- OO = "object oriented"
- in OO design we model a system as a collection of objects
 - e.g. a GUI comprises a desktop with windows which have panes, scrollbars, menus etc.
 - we think of this as a collection of particular types of objects
- an object
 - has its own state
 - has its own operations
 - for querying its state: **accessors**
 - for updating its state: **mutators**
 - for initialisation: **constructors**
 - can depend on and collaborate with other objects

Classification in OO Design

- classification is used to differentiate between different types of objects
- different types of objects belong to different classes
 - according to what information they contain
 - how they may change over time
- high level OO design is about
 - finding the classes
 - identifying the characteristic properties of those classes
 - identifying the relationships between different classes of objects

Class vs Object

- a class is a description, or specification of an object's structure and behaviour
 - a class is a design-time description
 - in OO implementation, the source code is just a collection of classes
- new objects can be built according to the model defined by a class
- we say: *an object is an instance of its class*
 - in design, objects represent real-world entities
 - either concrete: e.g. a person, a book, a library, a clock
 - or artificial / synthetic: e.g a compiler, a mathematical formula
 - in implementation, **objects are run-time entities** constructed during execution of a program
 - objects are records with their own data fields, which
 - know their own type or class, and
 - have access to the operations defined by their class

Object Technology

- OT is technology built around support for OO models
 - design notations
 - class diagrams
 - UML: unified modelling language
 - early OO programming languages
 - Simula'67, Smalltalk (early 80's), Eiffel and C++ (late 80's)
- dominant software technology in industry
 - became popular in 1990's, especially with Java
 - why Java?
 - small language, large library, portable across OS's
 - now most software applications are object-oriented
 - C++, Java, C#, VB
 - Python, Ruby, Groovy
 - older programming languages are often extended to support OO
 - OO Cobol, CLOS (from Lisp), OCaml (from ML)

Strengths of OO Techniques

- ***encapsulation*** of data + operations
 - combined definition of data + function in a class
 - easier to maintain data consistency and system integrity
- ***modularity***
 - distinct software components
 - separate development
 - separate compilation
 - separate deployment
 - modular designs
 - ***low coupling*** between components
 - limited and well-defined dependency between components
 - ***high cohesion*** within components
 - all features of a component are related

Strengths of OO Techniques

- ***abstraction***
 - clear separation between
 - **interfaces** needed to use components of a system
 - specification of functions and data types
 - **implementation** details
 - choice of data representation
 - code to implement functions
 - principle of information hiding
 - achieve modularity by limited exposure of implementation
 - interfaces define allowed "public" or external usage
- flexible code ***reuse***
 - code customisation via method overriding
 - support for application frameworks
 - *hooks* for application specific code
 - "don't call us, we'll call you"
 - assists standardisation of "look and feel"

Seamlessness of OO Techniques

- the object model can be used at all levels of software development
 - analysis and design
 - design time objects reflect real-world entities
 - OO can be used for modelling and simulating the world
 - whether the "world" is real or virtual
 - programming
 - runtime objects
 - either implement design time objects
 - emulating real-world entities
 - provide services to support the implementation
 - e.g. interface with operating system
 - OO can be used for building systems out of objects
 - the objects describe different types of building blocks, or components, for the system
 - overall system behaviour is determined by interactions amongst the objects

The non-OO Mindset

- older, *structured development*, techniques
 - use top-down function decomposition
 - design functions using lower level functions
 - bottom out in basic code
 - strong emphasis on control flow: what happens when
 - sequencing, branching, looping and recursion
 - data is structured: elements are grouped to satisfy functional descriptions at all levels
 - in C, all functions are top level
 - no nesting of function definitions
 - fails to reflect top-down structure
 - but makes functions more reusable
 - variables are either top level (global) or inside functions (local)
 - local variables are private to their function body
 - communication between functions is either via function parameters or global variables

The OO Mindset

- *OO development* is bottom-up
 - applications are composed of collaborating objects
 - there need not be just one top-level function
 - different objects are responsible for providing different classes of services for other objects to use
 - the state of the system evolves as objects are created and updated
 - emphasis is more on dependencies between objects
 - rather than the control flow for a particular method
 - individual behaviours (methods of objects may still be designed top-down
 - structured design applies at level of individual methods
- this scales better for large-scale applications
 - data-driven design tends to be more stable than function-driven design
 - OO aims to achieve effective reuse of both design and implementation

Basic OO Concepts: Objects and Classes

- object
 - encapsulation of data + operations
 - every object has a unique ID
 - logically equivalent to a fixed memory address
 - object identities are used to reference objects
 - information hiding
 - separation of public interface from internal representation
 - operations
 - methods
 - functions with a notion of current active object
 - calls, events, messages
- classes define objects
 - objects are *instances* of classes
 - objects of same class have common
 - data structure
 - methods
 - different objects of same class
 - have a different object identity
 - may have different data values

Introduction to Java

Example of a Class Definition

```
class Point
{
    double x;
    double y;

    void translate(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
}
```

Example of a Class Definition

```
class Point
{
    double x;
    double y;

    void translate(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
}
```

Example of a Class Definition

class header declares
name of the class

class body declares
members of the class

```
class Point
```

```
{  
    double x;  
    double y;  
  
    void translate(double dx, double dy)  
    {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```

Example of a Class Definition

```
class Point
```

class header declares
name of the class

class body declares
members of the class

```
{
```

```
double x;
```

```
double y;
```

x and y are **instance variables** or **fields**
of double precision floating point type

```
void translate(double dx, double dy)
```

```
{
```

```
    x = x + dx;
```

```
    y = y + dy;
```

```
}
```

```
}
```

Example of a Class Definition

```
class Point
{
    double x;
    double y;
    void translate(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
}
```

class header declares name of the class

class body declares members of the class

x and y are instance variables or fields of double precision floating point type

translate is a method with two input arguments of type double and void return type

Example of a Class Definition

```
class Point
{
    double x;
    double y;

    void translate(double dx, double dy)
    {
        x = x + dx;
        y = y + dy;
    }
}
```

class header declares name of the class

class body declares members of the class

x and y are instance variables or fields of double precision floating point type

translate is a method with two input arguments of type double and void return type

body of method increments fields of this Point object by an amount given by the method call's actual arguments

Construct and Move a Point

```
Point p = null;  
    // p == null
```

```
p = new Point();  
    // p.x == 0 && p.y == 0
```

```
p.translate(10,20);  
    // p.x == 10 && p.y == 20
```

```
p.translate(0,20);  
    // p.x == 10 && p.y == 40
```

Construct and Move a Point

```
Point p = null;  
    // p == null
```

```
p = new Point();  
    // p.x == 0 && p.y == 0  
p.translate(10,20);  
    // p.x == 10 && p.y == 20  
p.translate(0,20);  
    // p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

```
Point p = null;  
    // p == null
```

```
p = new Point();  
    // p.x == 0 && p.y == 0  
p.translate(10,20);  
    // p.x == 10 && p.y == 20  
p.translate(0,20);  
    // p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

initialisation of reference `p` does not refer to any `Point` object yet

```
Point p = null;  
// p == null
```

```
p = new Point();  
// p.x == 0 && p.y == 0  
p.translate(10,20);  
// p.x == 10 && p.y == 20  
p.translate(0,20);  
// p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

```
Point p = null;
```

initialisation of reference `p` does not refer to any `Point` object yet

```
// p == null
```

allocates a new `Point` object on the heap, initialised via the default constructor `Point()`. Returns a reference to the new `Point` object.

```
p = new Point();
```

```
// p.x == 0 && p.y == 0
```

```
p.translate(10,20);
```

```
// p.x == 10 && p.y == 20
```

```
p.translate(0,20);
```

```
// p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

```
Point p = null;
```

initialisation of reference `p` does not refer to any `Point` object yet

```
// p == null
```

allocates a new `Point` object on the heap, initialised via the default constructor `Point()`. Returns a reference to the new `Point` object.

```
p = new Point();
```

```
// p.x == 0 && p.y == 0
```

default initial values of a `Point p`'s fields

```
p.translate(10,20);
```

```
// p.x == 10 && p.y == 20
```

```
p.translate(0,20);
```

```
// p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

```
Point p = null;
```

initialisation of reference `p` does not refer to any `Point` object yet

```
// p == null
```

allocates a new `Point` object on the heap, initialised via the default constructor `Point()`. Returns a reference to the new `Point` object.

```
p = new Point();
```

```
// p.x == 0 && p.y == 0
```

default initial values of a `Point` `p`'s fields

```
p.translate(10, 20);
```

```
// p.x == 10 && p.y == 20
```

calls `translate` method for target object referenced by `p`

```
p.translate(0, 20);
```

```
// p.x == 10 && p.y == 40
```

declaration of a local variable `p` of type `Point` for holding a reference to a `Point` object

Construct and Move a Point

```
Point p = null;
```

initialisation of reference `p` does not refer to any `Point` object yet

```
// p == null
```

allocates a new `Point` object on the heap, initialised via the default constructor `Point()`. Returns a reference to the new `Point` object.

```
p = new Point();
```

```
// p.x == 0 && p.y == 0
```

default initial values of a `Point` `p`'s fields

```
p.translate(10, 20);
```

```
// p.x == 10 && p.y == 20
```

calls `translate` method for target object referenced by `p`

```
p.translate(0, 20);
```

```
// p.x == 10 && p.y == 40
```

updated values of `Point` `p`'s fields

Comparison with functions in C

- in Java, the translate method implicitly takes the target object as its first argument
- in C, you might declare

- `Point` as a typedef for a struct with `x` and `y` fields
- `translate` as a C function

```
void translate(Point *this, double dx, double dy) {  
    this->x = this->x + dx;  
    this->y = this->y + dy;  
}
```

- the first parameter `this` is a pointer to a `Point` struct
- the Java code for `translate` does not refer to this parameter
 - a Java call `p.translate(10,20)` automatically binds the `Point` object pointed to by `p` as the current instance of `Point`
 - the equivalent call in C would be `translate(p,10,20)`

Scope of method names in OO: dot-call syntax

- the form of the call `p.translate(10,20)` is called **dot-call syntax**
- the scope of method names extends outside the class where the methods are defined
 - two methods defined in different classes with the same name do not clash
 - they are differentiated by the type of their target in the dot-call: the target is the expression before the dot
 - given a `book` that refers to a `Book` object, a call `book.translate(10,20)` might invoke a translation into French for pages 10 to 20

Java Development Tools

- on your own machines
 - download from Sun
 - you want Java SDK 6 SE (standard edition)
 - for desktop applications
 - don't need enterprise edition (EE) for server applications or mobile edition (ME)
 - install Sun's JDK 6 (aka JDK 1.6)
 - including Java 6 API libraries
 - recommend
 - download full documentation (for tools and library API)
 - download library source
 - browsing library source (in eclipse) is a great way to see how professional Java programmers code
- available in CSE labs
 - **always check on CSE machines** before submitting for lab assignments and project work

Checklist for Java SDK 6

- **bin** directory
 - **javac**
 - compiler
 - generates `.java` files from `.class` files
 - **java**
 - executes `.class` files
 - **javap**
 - disassembler
 - use `-c` option to generate readable bytecodes
 - **javadoc**
 - generates html documentation from `.class` files
 - special javadoc comments in `.java` files
 - embedded HTML is recognised
- tool options
 - use `-d` to set target directory if not current
 - use classpath option or set `CLASSPATH` environment variable to locate any `.class` files needed for import
- online documentation with JDK:
 - Java library (API) reference
 - Java tools documentation
 - how to set `CLASSPATH`
 - also see *First Cup of Java* trail in Java Tutorial
- select library API source for download
 - allows you to browse library source (in eclipse)
 - a great way to see code by professional Java programmers
- separate online tutorials
 - *Your First Cup of Java*
 - *Getting Started*
 - *Learning the Java Language*

Review

- notion of design quality
- software design
- OO Concepts
 - classes describe objects
 - in OO programming, objects are created according to the blueprint defined by their class
- introduction to Java
 - a class defines fields (data) + methods (operations)
 - compiler checks code and generates bytecodes (class files)
 - JVM executes programs by interpreting and/or compiling bytecodes

Java Execution Model: JVM

- Java defines a standard computer for executing its programs
 - called the *Java Virtual Machine* or JVM
 - JVM is a stack-based machine
 - with no registers
- the JVM is implemented on different real platforms to present a uniform programming model
 - this makes Java programs highly portable
 - hence the Java catch-phrase: *write once, run everywhere*
 - portable code idea adopted from earlier languages
 - Smalltalk
 - p-code for Pascal

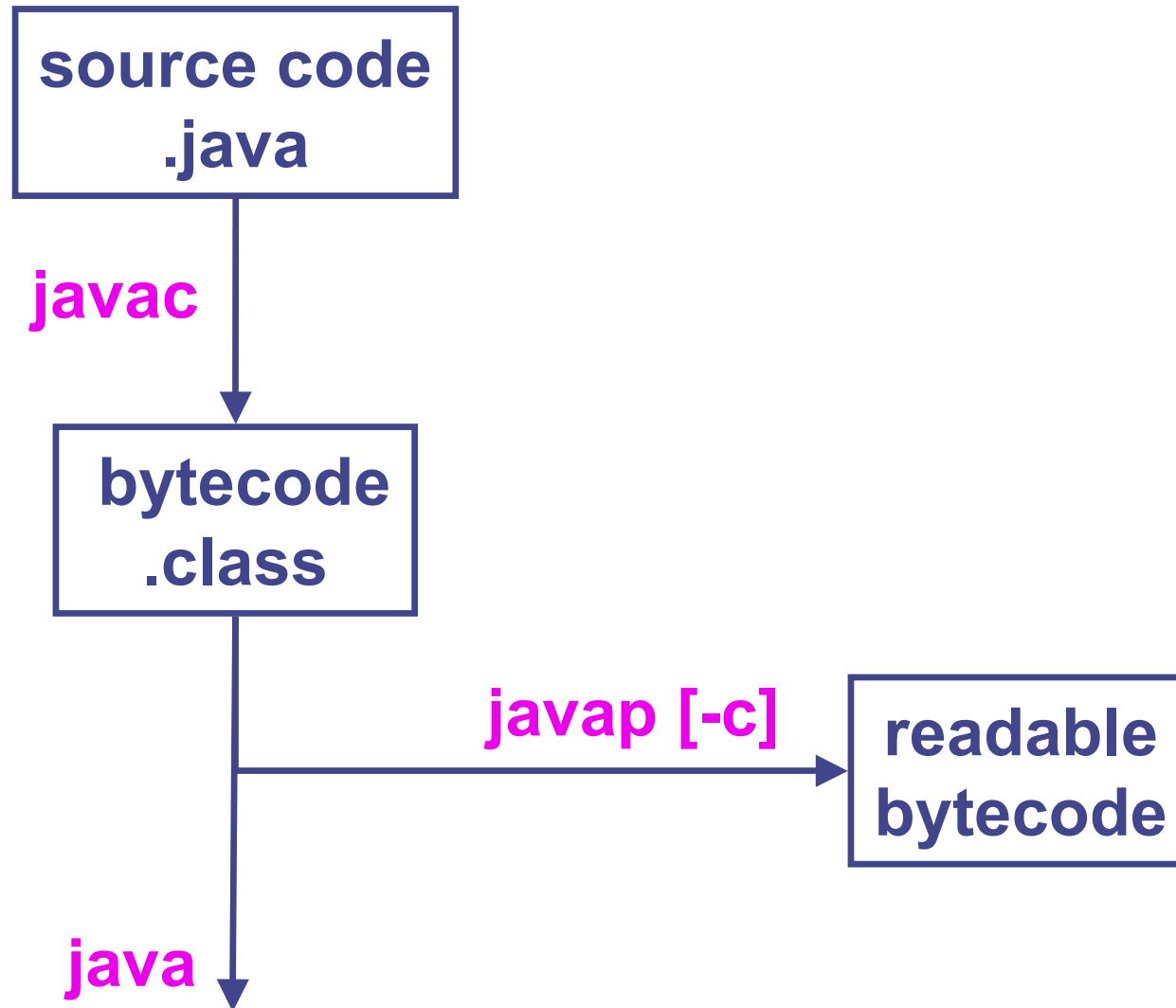
Running the JVM

- the command: `java XXX` runs the JVM
 - the JVM reads instructions from the input file `XXX.class`
 - note: `XXX` is just a placeholder for a real filename
 - the `XXX.class` file must be in Java bytecode format
 - by convention, a ".class" file refers to Java bytecodes
 - the `XXX.class` file must have a main method
- bytecodes are an *intermediate language*
 - intermediate between Java source code and executable machine code
 - early JVMs simply interpreted these bytecodes and did not generate native executable code
 - current JVMs are JIT (*just-in-time*) compilers
 - generate executable code on the fly
 - superior runtime performance

Compiling Java

- Java source code occurs in ".java" files
- the command: `javac XXX.java`
 - parses and type checks the file XXX.java
 - if no errors, generates bytecode file XXX.class
- compilation of multiple files is possible
 - e.g. `javac *.java`
 - will compile multiple bytecode files
 - any number of these may have a main method
 - compiler can resolve code dependencies between files
- normally, a file called XXX.java should contain source code defining a Java class called XXX
 - there should be a declaration `class XXX { ... }` in the file

In Summary



Hello World

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

- this defines a public class `Hello` with a single public method `main`
 - `public` means it can be called from other classes
 - `static` means the method belongs to the class and not to any current object instance
 - this is more like a C function, which is qualified by the class name
- when invoked as the main method from the command line the array `args` of `Strings` picks up any extra command line arguments
 - note that arrays should be declared with the array type constructor `[]` qualifying the array element type, not the data variable
 - C style array declarations used to be allowed, but are now deprecated
- `System.out` refers to standard output via the static variable `out` of class `System` and `println` is a standard method for output streams

Java Execution Model

- recall from last lecture
 - the command: `java XXX` runs the JVM
- roughly speaking, this
 - starts the JVM
 - loads the class files found via `XXX.class`
 - loads all supplier classes as well
 - initialises classes and runs static initialisation blocks
 - creates the main Thread
 - calls `XXX.main` on this Thread
 - NB JVM is multithreaded, but in this course we will restrict ourselves to single-threaded code

Calls are made on the Stack

- every method call creates a new call frame on the current thread
 - this frame is "pushed" onto the call stack
 - when a call returns, the frame is "popped"
- simple method code is single entry, single exit
 - structured programming discipline
 - makes programs easier to reason about
 - avoids spaghetti code with goto's with arbitrary control flow
- the execution model is actually not so simple
 - exceptions yield an alternative flow of control
 - BUT with *structured exception handling*, control flow is not so simple
 - failing computations can cause the stack to unwind too
 - more on exceptions later!

Static Methods

- a **static** method belongs to a class
 - instead of an object instance
 - static methods cannot refer to the current instance of the class
 - that is, static methods cannot use **this**
 - or any other instance variables (i.e. non-static fields) of the class
- static methods are just like functions in the C language
 - *except* outside of the class where the method is defined, any use of the method name must be qualified by the classname
 - unless that name is explicitly brought into scope via a static import declaration (more on imports later)
- in Java, calls to static methods evaluate the arguments and construct a new call frame on the stack
 - binding the values of the actual arguments to the formal parameters of the method definition
- demo of stack execution trace: **MinimiseUtils.min3**

Overloaded Methods

- different methods in the same class are *overloaded* if they have the same name
- overloaded methods must have different numbers and/or types of arguments
 - so a call with given method name can determine which method is intended
- see Java API documentation for `java.io.PrintStream`
 - `void println()`
 - `void println(int)`
 - `void println(Object)`
 - `void println(String)`
 - ... one println method for each primitive Java type

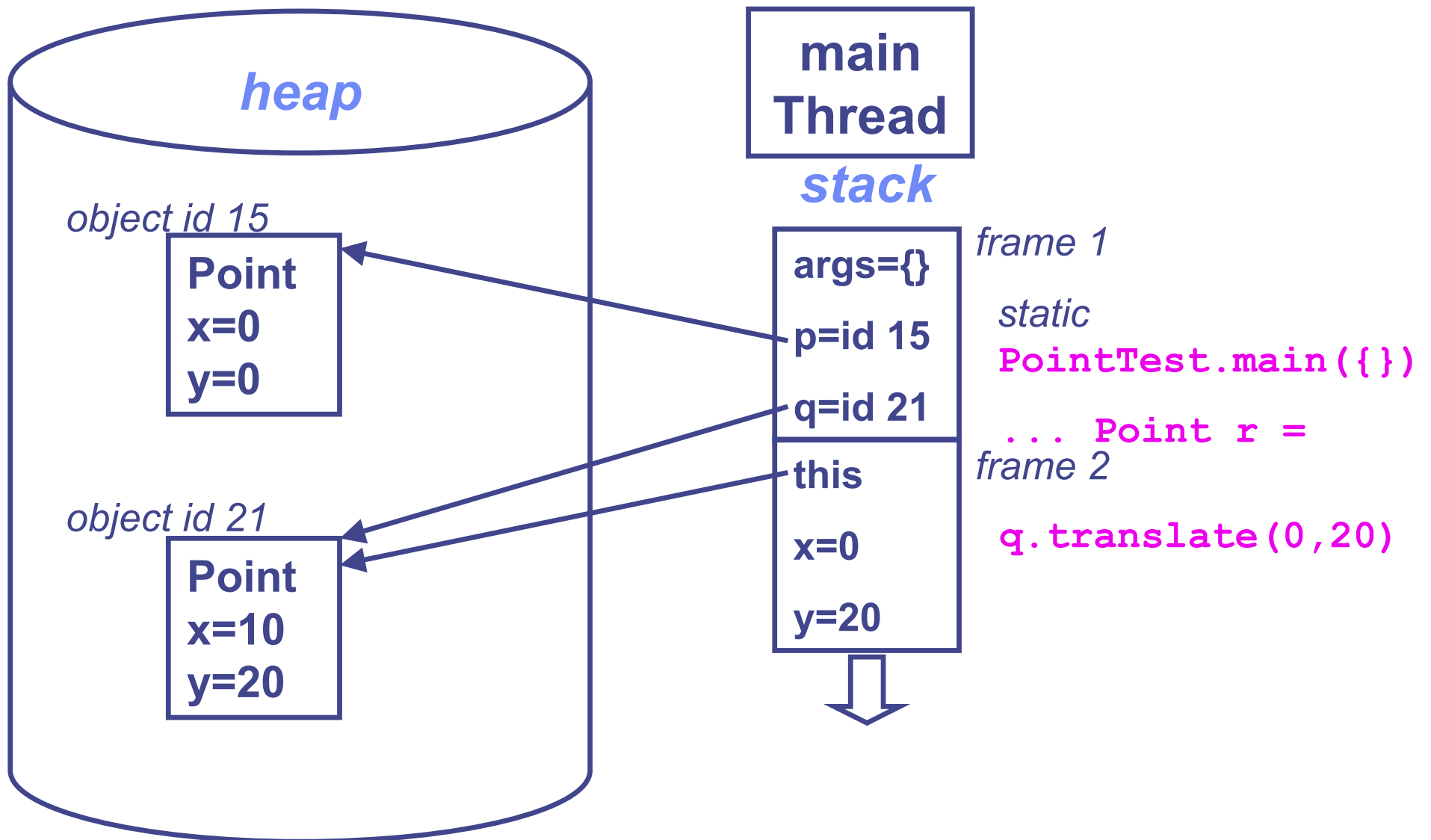
Non-static Methods

- methods are non-static **by default**
- they must be called on some *target object*
 - using dot-call notation
 - in the call: **p.translate(10,20)**
 - **p** refers to the target object
- in the call, the actual target **p** is bound to the local parameter **this** in the method body
 - this is effectively the first parameter of the call

Objects are made on the Heap

- objects are created as a result of executing a **new** expression of the form **new ClassName (actual args ...)**
 - memory is allocated for the object in the **heap**
 - *memory heaps and heap ADTs are different beasts*
 - *don't confuse the different uses of the word "heap"*
 - all object live in the heap in Java
- every class named ClassName has one or more **constructor methods** with the same name
 - constructors may be **overloaded**
 - if no explicit constructor is defined
 - a default constructor, with no args, CCC() is automatically provided by the compiler
- constructors have no return type
 - they are only used in new expressions

Calls on Stack; Objects on Heap



Access Modifiers

- *information hiding* reduces dependency between different parts of our code
 - when we make changes to the code, we protect ourselves from having to make changes everywhere
- classes may be declared public or not
- class members may have access modifiers
 - **public**, **private**, **protected**, or *default*
- access modifiers control where those entities can be named
 - **public** entities are accessible from anywhere
 - **private** members only be named within the same class
 - *default* with no modifier, access is restricted to code in the same package
 - more on packages later
 - **protected** access is related to class extension and inheritance
 - more on inheritance later

Point Example Extended

- the previous **Point** example is extended here
- x, y fields are private to protect them from modification by external clients
 - **getX** and **getY** methods allow us to inspect the values of the x, y fields
 - but we cannot modify them
- return type of translate method changed from **void** to **Point**
 - **translate** is no longer a mutator method
 - it now queries the current point, and returns a new one, applying the required translation
- so external clients now have no way to change the state of a **Point**
 - such an object is called **immutable**
- constructor **Point(x,y)** has been added
 - note that method arguments "hide" the fields with the same names
 - the field names can still be accessed via a reference to **this**
- constructor **Point()** has been added
 - default initial values are left as is
- added separate **PointTest** code
- **NOTE:** all code appears within a class in Java

Execution Tracing in Debugger

- eclipse provides debugging facilities
- either set breakpoints in code
 - double click on left side of code editor
 - click Debug button
- or
 - select Debug As ...
 - select Stop in main option
- Debug perspective then allows you to step through or over code
 - inspecting variables via stack frames
 - can access objects on the heap from the stack
 - selecting an object id will display the result of `toString()`

Summary

- *method calls* in Java push frames onto the *stack* of the current thread
 - instance methods bind **this** to the target object of the call
 - method returns pop the frame from the stack
- *new expressions* allocate new objects on the *heap*
- *static* methods have no current instance **this**
- *overloaded methods* are methods in same class, with same name, but with different argument types
- *constructor methods* initialise objects, and are only callable within a new expression
- information hiding: *private* members cannot be directly accessed outside a class; *public* members can be accessed from anywhere