MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Programming Paradigms
159.272
Java Language Basics

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

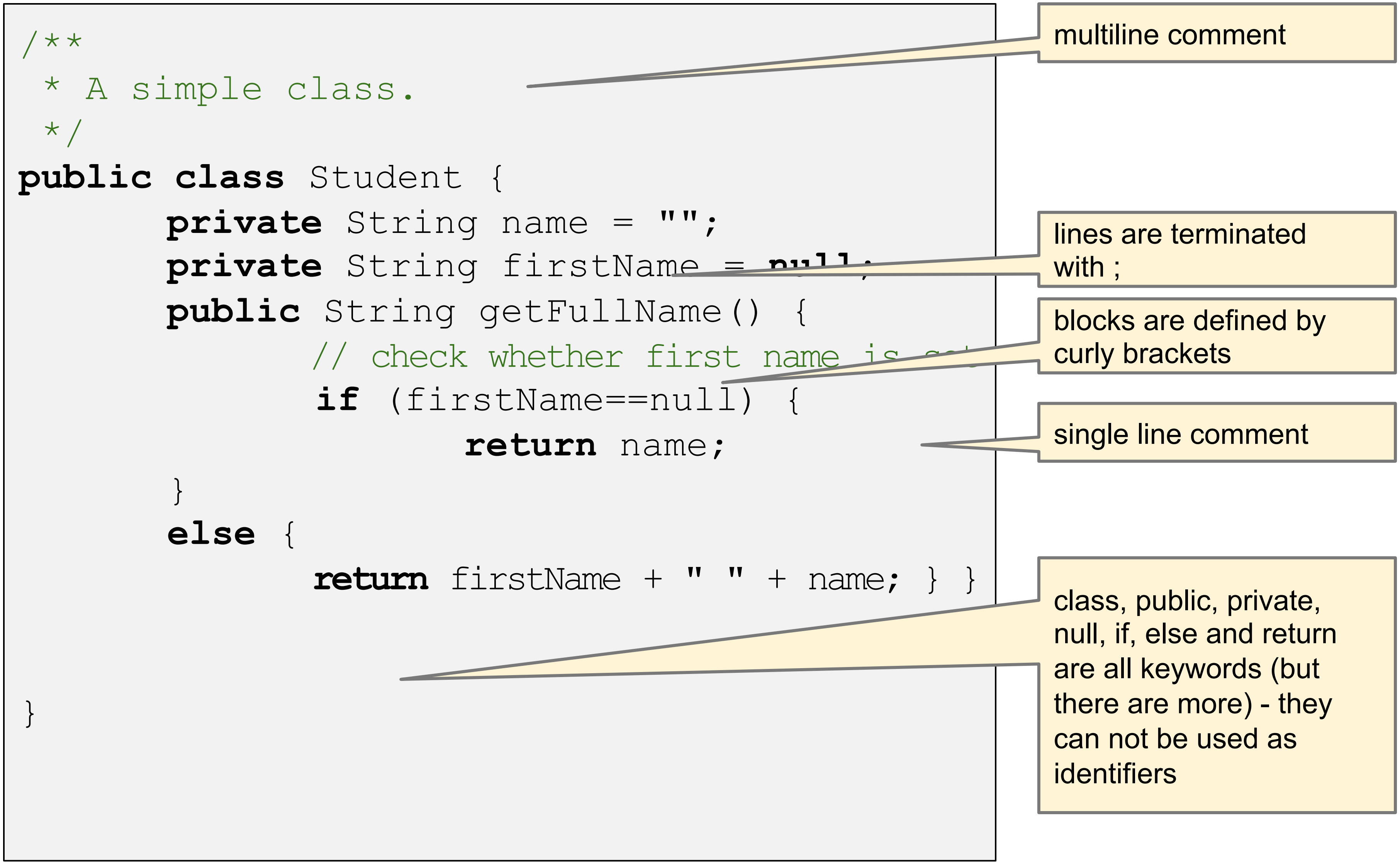
Readings

1. The Java Language Specification, Chapter 4: Types Values and Variables.
   http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.htm
2. Java Tutorial, trail: Learning the Java Language
   http://docs.oracle.com/javase/tutorial/java/index.html

Overview

- Java syntax 101
- classes and objects
- objects first vs classes first
- calling methods
- primitive types
- boxing
- arrays
- enums
- type checking
- object identity
- mutability
- strings

Java Syntax 101



```
/**
 * A simple class.
 */
public class Student {
       private String name = "";
       private String firstName = null;
       public String getFullName() {
             // check whether first name is set
             if (firstName==null) {
                    return name;
       }
       else {
             return firstName + " " + name; } }



}
```

multiline comment

lines are terminated with ;

blocks are defined by curly brackets

single line comment

class, public, private, null, if, else and return are all keywords (but there are more) - they can not be used as identifiers

Java Syntax 101 ctd

- Java uses C-like syntax (like C, C++, C#, ObjectiveC, JavaScript, ..)
- identifiers (class, method names etc) can use alphanumeric characters, the first character can not be numeric: `foo42` is ok, but not `42foo`
- identifiers can also contain underscores '_', as in `foo_42` or `_foo`
- keywords cannot be used as identifiers
- Java is case sensitive

Java Syntax 101 - Conventions

- conventions are rules are not enforced by the compiler, but widely accepted within the community
- some tools depend on these conventions (aka "convention over configuration")
- Class names should start with an uppercase character
- method names start with a lowercase character
- field names start with a lowercase character, except static fields - they use uppercase characters and _
- compound names are written in "CamelCase":
  - example: `PartTimeStudent` (and not `Parttimestudent` or `Part_Time_Student`)

Classes

- Java is a **class-first** language - like most other OO languages

- to create objects, we first need to define a **class** for this object

- once a class has been defined, instances can be created by using **constructors**

- there can be multiple constructors with different sets of parameters - this is called **constructor overloading**

A Simple Class

```
public class Student {
        // (multiple) constructors
        public Student() {}
        public Student(String fName, String lName, int a) {
                firstName = fName;
                lastName = lName;
                age = a;
        }
        // state: instance variables
        public String lastName = "";
        public String firstName = "";
        public int age = 0;
        // behavior: methods
        public String getFullName() {
                return firstName + " " + lastName;
        }
}
```

two constructors

state: instance variables (aka properties or fields)

behavior: methods

Instantiation: Using a Class

```
public class Test1 {

        public static void main(String[] args) {
                Student aStudent = new Student();
                aStudent.firstName = "John";
                aStudent.lastName = "Smith";
                aStudent.age = 20;

        System.out.println(aStudent.getFullName());
        }

}
```

create a new object of the type Student by invoking the constructor

changing the state of this object

use a method of Student to print the full name

Alternative Instantiation

create a new object of the type Student by invoking the constructor, initialise state through parameters

```
public class Test2 {

       public static void main(String[] args) {
             Student aStudent = new Student("John","Smith",20);
             System.out.println(aStudent.getFullName());
       }

}
```

Objects First

- an **objects first** approach is also possible, but not supported by Java
- however, **JavaScript** uses objects first
- objects can be directly defined as object literals
- then prototypes can be used to reuse functions defined in objects

Objects First with JavaScript

```
var aStudent = {
    firstName: "John",
    lastName: "Smith",
    age: 20,
    getFullName: function () {
        return this.firstName + " " + this.lastName;
    }
};
```

create a new object with state and behaviour, no class necessary.
This is also called an "object literal".

Note: *JavaScript is not Java, this is a completely different language !*

Objects First with JavaScript ctd

- in JavaScript, properties and functions can be added dynamically to objects
- example: `aStudent.isMale = function() {..}`
- when creating many objects of the same kind, the same functions have to be created and added again and again - this is very wasteful
- this can be solved by attaching functions to the prototype object each object has
- the prototype is the counterpart of a class in a class first language
- see http://ejohn.org/apps/learn/#64 for more info

Classes as Types

classes are used as types when:

- declaring variables:

```
String lastName
```

- declaring method return types:

```
String getLastName()
```

- declaring method and constructor parameter types:

```
Student(String s1,String s2,int i)
```

Java Types (overview)

- classes are so-called **reference types**
- other reference types are enumerations and interfaces (to be discussed later)
- Java also has 8 built-in primitive (non-reference) types
- primitive types are also known as **value types**

Primitive Types

- boolean - two possible values, with literals true and false
- **char** - 16bit characters
- Numbers (**int** types):
  int : 32bit int
  byte: 8bit int
  short: 16bit
  long - 64bit
- Numbers (**float** types)
  float: 32bit floating point number
  double: 64bit floating point numbers

Reference Types

```
public class TestReferenceTypes {

      public static void main(String[] args) {
            Student aStudent = new
Student("John","Smith",20);
            resetName(aStudent);
            System.out.println(aStudent.lastName);
      }
      public static void resetName (Student s) {
            s.lastName = "?";
      }
}
```

Reference Types

```
public class TestReferenceTypes {

      public static void main(String[] args) {
            Student aStudent = new
Student("John","Smith",20);
            resetName(aStudent);
            System.out.println(aStudent.lastName);
      }

      public static void resetName (Student s) {
            s.lastName = "?";
      }
}
```

a reference to the actual object is passed to resetName and the state o the object is changed

- this will print "?"
- this is sometimes called **reference leaking -** the reference to an object is leaked to another method or class and can be modified

Value Types

```
public class TestValueTypes {

      public static void main(String[] args) {
            Student aStudent = new
Student("John","Smith",20);
            resetAge(aStudent.age);
            System.out.println(aStudent.age);
      }
      public static void resetAge (int age) {
            age = 18;
      }
}
```

Value Types

```
public class TestValueTypes {

        public static void main(String[] args) {
                Student aStudent = new
Student("John","Smith",20);
                resetAge(aStudent.age);
                System.out.println(aStudent.age);
        }
        public static void resetAge (int age) {
                age = 18;
        }
}
```

- this will print **20**
- the value is passed to `resetAge`, it is disconnected from the `age` property of `aStudent`

If Numbers were Objects ..

- in languages like Smalltalk and Ruby, numbers are objects
- this can make the type system simpler
- then code like this is possible (this is Ruby): `3.odd?` - or in Javaish syntax: `3.isOdd()`

Static vs Dynamic Typing

- in Java, each variable has to be declared using a type
- this makes Java a **statically typed language**
- on the other hand, in a **dynamically typed language** variables are not associated with a fixed type in the source code (at compilation time)
- example: in JavaScript, variables can be declared as `var x = ..` (`var` is not a type, but a general declaration keyword for variables)
- Python allow dynamic typing : `x =` ...

Strong vs Weak Typing

- Java is (relatively) **strongly typed**: methods are only compiled and executed if the correct types are used

- **weakly typed** languages perform implicit **type conversions**

- this is sometimes convenient, but can make the language unpredictable

Type Conversion Example

- in most modern languages + has a different meaning depending on the type:
    - for integers, it means simply "add"
    - for strings, it means concatenate (`"a"+"b"` yields `"ab"`)
    - this is called **operator overloading**

- now consider `5+"37"` .. which + should be used?

Type Conversion Example ctd

- a weakly typed language would try to either*:
  - convert the integer `5` to `"5"` - this would yield `"537"` (example: JavaScript)
  - or convert the string `"37"` to the integer `37`, this would yield `42` (example VisualBasic)
- although Java is generally strongly typed, it has some elements of weak typing as well:
  - if one operand of `+` is a string, it will convert the the second one to a string
  - i.e., all expressions `"1"+"2"`, `1+"2"` and `"1"+2` evaluate to `"12"`, while `1+2` evaluates to `3`
  - see also https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.4

* example from: http://en.wikipedia.org/wiki/Dynamic_typing#Strong_and_weak_typing

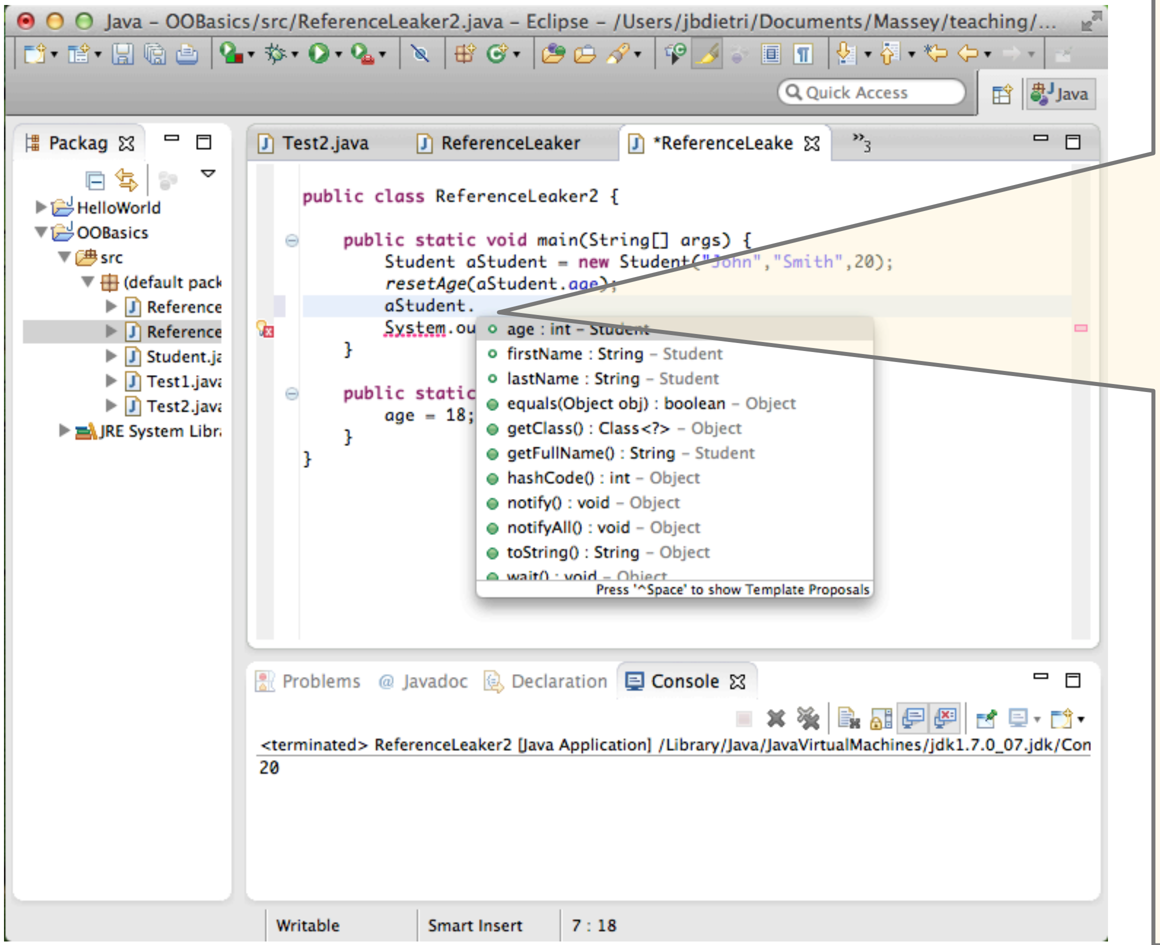
Advantages of Static Typing

- with static typing, the compiler can perform many checks to find out whether code is correct

- the compiler enforces type consistency

- static typing facilitates autocompletion style tooling in IDEs

Using Static Typing: Autocomplete



```
public class ReferenceLeaker2 {

    public static void main(String[] args) {
        Student aStudent = new Student("John","Smith",20);
        resetAge(aStudent.age);
        aStudent.
        System.ou
    }

    public static
        age = 18;
    }
}
```

```
age : int – Student
firstName : String – Student
lastName : String – Student
equals(Object obj) : boolean – Object
getClass() : Class<?> – Object
getFullName() : String – Student
hashCode() : int – Object
notify() : void – Object
notifyAll() : void – Object
toString() : String – Object
wait() : void – Object
                    Press '^Space' to show Template Proposals
```

```
<terminated> ReferenceLeaker2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_07.jdk/Con
20
```

after typing "aStudent." (dot) - the IDE (Eclipse) knows that aStudent is of the type Student, inspects the Student class to find all methods and fields that can be used, and displays them in a pop - up list.
This facilitates a dialog style communication between the programmer and the IDE.

Type Signatures

- methods have **return and parameter types**
- the list of parameter types can be empty
- **void** is used to indicate that a method returns no result
- this is called the **type signature** of a method
- exceptions (discussed late in the course) are also part of the signature
- note that the actual implementation of a method ("method body") is not part of the type signature
- Examples:
    - `int getTransparency(Color)`
    - `Date getDOB()`
    - `double average (int[])`
    - `void save()`

Type Checking by the Java Compiler

```
public int sum(int x,int y) {
     return "42";
}
```

Type Checking by the Java Compiler

```
public int sum(int x,int y) {
     return "42";
}
```

the Java compiler **will not compile** this as types are not consistent

the declared return type is an int, but the actual return value is a string !

Type Checking by the Java Compiler (ctd)

```java
// computes and returns the sum of two ints
public int sum(int x,int y) {
     return x-y;
}
```

Type Checking by the Java Compiler (ctd)

```
// computes and returns the sum of two ints
public int sum(int x,int y) {
     return x-y;
}
```

the Java compiler **will compile** this as the types are consistent

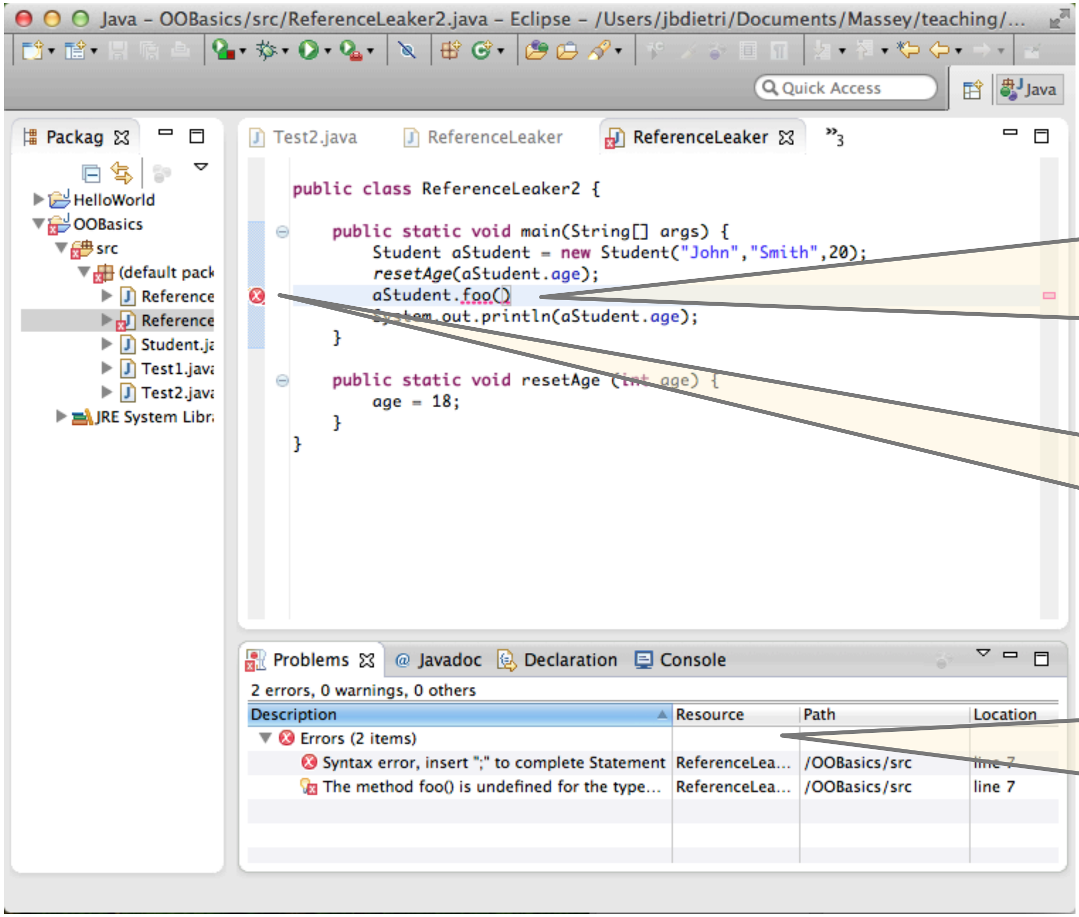
the compiler cannot understand the meaning (semantics) of the method

other tools and methods are needed to check code for this kind of violation

this will be discussed later (topic "semantics")

"if it compiles then it works" is wrong !!

Type Checking in Eclipse



```
public class ReferenceLeaker2 {

    public static void main(String[] args) {
        Student aStudent = new Student("John","Smith",20);
        resetAge(aStudent.age);
        aStudent.foo()
        System.out.println(aStudent.age);
    }

    public static void resetAge (int age) {
        age = 18;
    }
}
```

2 errors, 0 warnings, 0 others

Description	Resource	Path	Location
Errors (2 items)			
Syntax error, insert ";" to complete Statement	ReferenceLea...	/OOBasics/src	line 7
The method foo() is undefined for the type...	ReferenceLea...	/OOBasics/src	line 7

the compiler knows that there is no method foo() in Student, and highlights this as error

markers will be added to the sidebar to tag errors

all located errors are added to a list that can be displayed in the Problems view

Hint: hover the mouse cursor of the error, and Eclipse will display a list of quick fixes, e.g. methods with similar names

Boxing

the following code can be compiled and executed (and it prints 42 to the console, as expected):

```
public class TestBoxing {

        public static void main(String[] args) {
                printToConsole(42);
        }

        public static void printToConsole (Object object) {
                System.out.println(object);
        }
}
```

printToConsole has Object (the most general **reference type**) as parameter, but is called with a **value type** (int)

Boxing ctd

- the Java compiler does some magic here: it turns the int value into an object by **wrapping** (**boxing**) is with an instance of `java.lang.Integer`
- such **wrapper classes** exist for all value types:
    - `java.lang.Boolean`
    - `java.lang.Float`
    - ...
- this is convenient, in particular when working with **lists** and similar data structures
- this can be seen as an element of **weak typing** in Java

Comparing Objects and Values

- the built-in == operator can be used to compare objects
- it returns true if two references represent the same object, false otherwise
- for value types, == compares the actual values
- the opposite of == is !=
- != means "not the same as"

Comparing Objects - Example 1

```
int x = 4;
int y = 38;
System.out.println(42 == (x+y))
```

Comparing Objects - Example 1

```
int x = 4;
int y = 38;
System.out.println(42 == (x+y))
```

this will print **true** - the expressions are evaluated (x+y), and then the values will be compared

Comparing Objects - Example 2

```
Student aStudent = new Student("John","Smith",20);
System.out.println(aStudent == 42);
```

Comparing Objects - Example 2

```
Student aStudent = new Student("John","Smith",20);
System.out.println(aStudent == 42);
```

this causes a compilation error - reference and value types cannot be compared ("comparing apples and oranges")

Comparing Objects - Example 3

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent = aStudent;
System.out.println(anotherStudent == aStudent);
```

Comparing Objects - Example 3

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent = aStudent;
System.out.println(anotherStudent == aStudent);
```

this will print **true** - `aStudent` and `anotherStudent` both still reference the same object,
Note : only one object has been created !

Comparing Objects - Example 4

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent = modify(aStudent);
System.out.println(anotherStudent == aStudent);
..
Student modify(Student s) {
      s.age = 22;
      return s;
}
```

Comparing Objects - Example 4

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent = modify(aStudent);
System.out.println(anotherStudent == aStudent);
..
Student modify(Student s) {
      s.age = 22;
      return s;
}
```

- this will print **true** - `aStudent` and `anotherStudent` both still reference the same object, even though the state of this object has been changed by invoking modify
- only one object has been created !

Comparing Objects - Example 5

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent =
   new Student("Tim","Taylor",23);
System.out.println(anotherStudent == aStudent);
```

Comparing Objects - Example 5

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent =
                   new Student("Tim","Taylor",23);
System.out.println(anotherStudent == aStudent);
```

this will print **false** - `aStudent` and `anotherStudent` reference different objects, two objects have been created !

Comparing Objects - Example 6

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent =
                   new Student("John","Smith",20);
System.out.println(anotherStudent == aStudent);
```

Comparing Objects - Example 6

```
Student aStudent = new Student("John","Smith",20);
Student anotherStudent =
                   new Student("John","Smith",20);
System.out.println(anotherStudent == aStudent);
```

- this will still print **false** - `aStudent` and `anotherStudent` reference different objects, two objects have been created !
- Note: both objects have the same state (their instance variables have the same value)
- these objects are "twin-like" - equal but not identical

Identity vs Equality

- to express that two objects are equal, an **`equals`** method is used in Java that has to be implemented for each class

- the signature of this method is:
  `boolean equals(Object)`

- semantics (=meaning) of `equals` (informal): compare the state of the object on which equals is invoked with the object passed as parameter, and return true if the state of the two objects is the same, and false otherwise

Null

- `null` is a keyword in Java
- it can be used as a value for reference types
- `null` itself has no type
- `null` stands for "not initialised”

- if the value of a reference is null, and a method is called, a `NullPointerException` (a kind of error) is raised when the code is executed (not compiled!) - this will be discussed later

- there is only one null value - i.e., comparing multiple `null` references yields always `true`

Null - Example 1

```
int x = 42;
x = null;
```

Null - Example 1

```
int x = 42;
x = null;
```

this causes a compilation error – `Null` can only be used for reference types, but x is declared using a value type

Null - Example 2

```
Student aStudent = null;
aStudent.getFullName();
```

Assuming that `getFullName()` is defined in `Student`

Null - Example 2

```
Student aStudent = null;
aStudent.getFullName();
```

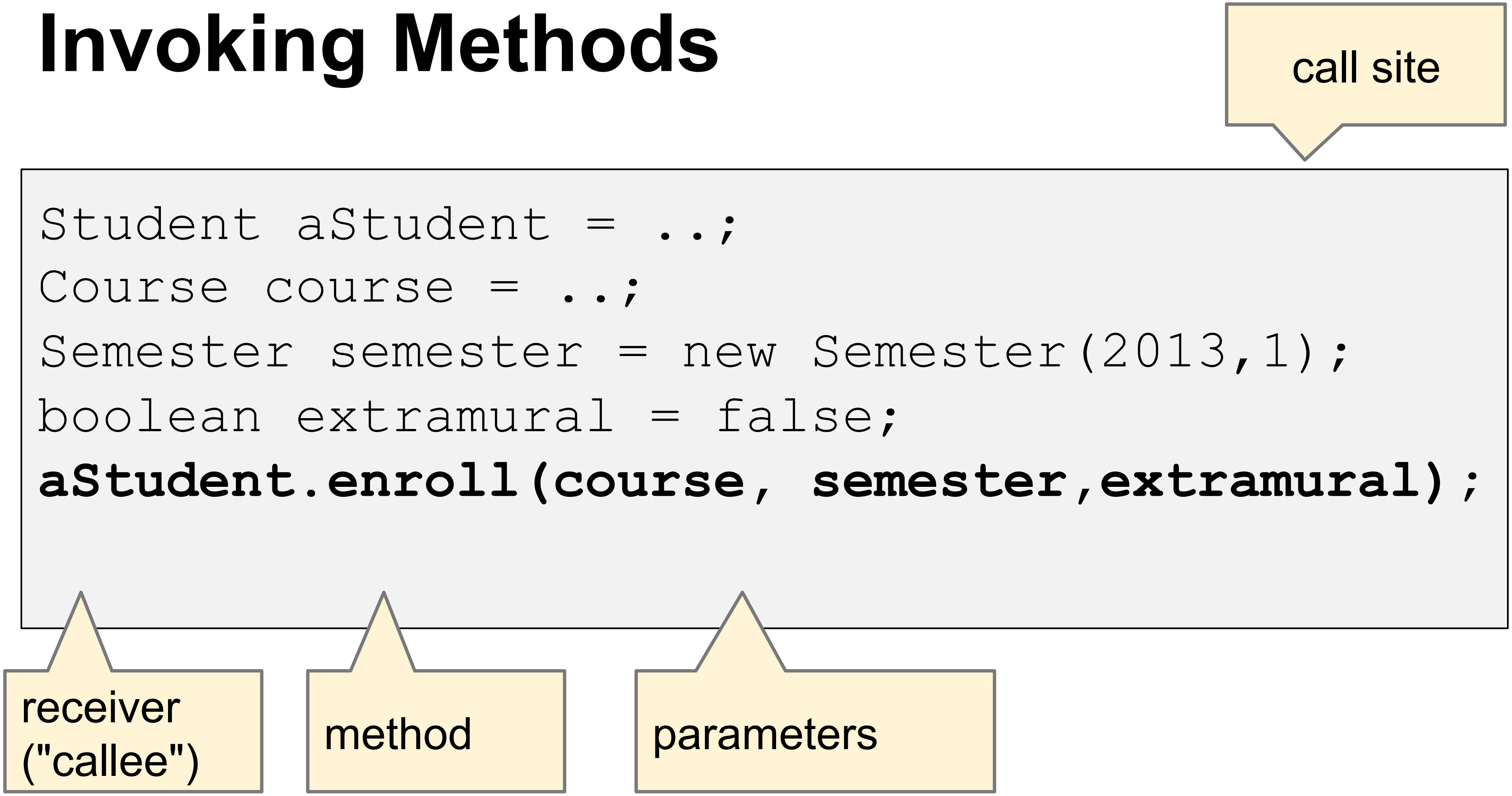
this can be compiled, but causes a `NullPointerException` when executed - there is no object on which `getFullName()` can be invoked !

Null - Example 3

```
Student aStudent = ..;
if (aStudent != null) {
   aStudent.getFullName();
}
```

- if it is anticipated that an object reference could become null, a **guard condition** can be used
- note that != is the negation of == (not equal)

Invoking Methods



call site

```
Student aStudent = ..;
Course course = ..;
Semester semester = new Semester(2013,1);
boolean extramural = false;
aStudent.enroll(course, semester,extramural);
```

receiver
("callee")

method

parameters

this

```
public class EnrollmentTest {
        public void printIfEligable(Student s) {
                if (isOldEnough(s)) {
                System.out.println(s.getFullName(
        }
    }
    void isOldEnough(Student s) {
                return s.age>=18
    }
}
```

where is the callee ?

- note that at there is an object if this code is executed = the object that has received `printIfEligable`
- this object is also the callee for the invocation of `isOldEnough`
- it is possible to make this explicit using the **`this`** reference

this (ctd)

```
public class EnrollmentTest {
       public void printIfEligable(Student s) {
             if (this.isOldEnough(s)) {
             System.out.println(s.getFullName());
       }
   }
   boolean isOldEnough(Student s) {
             return s.age>=18
   }
}
```

this is equivalent to the code on the previous slide

- it is recommended to use this (even though it is optional)
- this makes code more concise
- note that typing "**this.**" triggers autocompletion in most IDEs

- Note: you can’t use **this** within a static method (explained next!).

Static Fields

- fields (properties) can be declared as static
- this means that the value of the fields is shared across all instances of the respective class
- this is often used to add some "global" state

- for static fields, a special naming convention is often used: names consist of uppercase characters, and underscores to separate tokens in compound names

example:

```
   static String DEFAULT_COLOR_NAME = "gray";
```

Static Fields - Example 1

```
public class AClass {
       public int iField = 42;
       public static String S_FIELD = "Hello";
}
```

```
AClass object1 = new AClass();
AClass object2 = new AClass();
object1.iField = 0;
object1.S_FIELD = "Hola";
System.out.println(object2.iField);
```

Static Fields - Example 1

```
public class AClass {
      public int iField = 42;
      public static String S_FIELD = "Hello";
}
```

```
AClass object1 = new AClass();
AClass object2 = new AClass();
object1.iField = 0;
object1.S_FIELD = "Hola";
System.out.println(object2.iField);
```

- prints 42 - the value of iField in object2 has not been changed!
- only object2 was modified!

Static Fields - Example 2

```
public class AClass {
       public int iField = 42;
       public static String S_FIELD = "Hello";
}
```

```
AClass object1 = new AClass();
AClass object2 = new AClass();
object1.iField = 0;
object1.S_FIELD = "Hola";
System.out.println(object2.S_FIELD);
```

Static Fields - Example 2

```
public class AClass {
      public int iField = 42;
      public static String S_FIELD = "Hello";
}
```

```
AClass object1 = new AClass();
AClass object2 = new AClass();
object1.iField = 0;
object1.S_FIELD = "Hola";
System.out.println(object2.S_FIELD);
```

- prints "Hola" - the new value of `S_FIELD` applies to all instances of `AClass`

Static Field Access

```
public class AClass {
       public int iField = 42;
       public static String S_FIELD = "Hello";
}
```

```
AClass object1 = new AClass();
AClass object2 = new AClass();
object1.iField = 0;
AClass.S_FIELD = "Hola";
System.out.println(AClass.S_FIELD);
```

- to emphasise that static fields are not owned by a particular instance, an alternative syntax can (should) be used
- the field is accessed directly using the class
- note that this does not require any instance of this class to exist!

System.out

- this is a reference to an object usually representing the console
- the type of this object is `java.io.PrintStream`
- out is a **static variable** in `java.lang.System`
- typical use: console printing:

  `System.out.println("Hello World");`
- similar: `System.err` - usually used to log exceptions that occur when a program is executed

Static Methods

```
public class StudentPrinter {
        public static void main(String[] args) {
                Student aStudent = new Student();
                aStudent.firstName = "John";
                aStudent.lastName = "Smith";
                aStudent.age = 20;
                System.out.println(aStudent.getFullName());
        }
}
```

- there are also static methods
- these methods can also be invoked on the class directly - the class can be used as the callee
- note that static methods can not use the **this** reference - there is no current object
- example: the main method that makes Java classes executable is static

Final Fields and Methods

- both fields and methods can also be declared as **final**
- the value of final fields cannot be modified after the fields have been initialised
- final methods cannot be overridden (to be discussed later)

```
public class AClass {
       final int iField = 42;
       final String S_FIELD = "Hello";
}
```

Arrays

- Java arrays have a fixed length
- arrays of any type (reference and value) can be defined
- arrays can be multidimensional
- example:
  - int[] numbers = new int[42] // an array of 42 integers
  - String[] numbers = new String[42]
  - String[][] numbers = new String[3][3]
- access is with [<index>], **the first index is 0 !!**
- if an index outside the range is accessed, a runtime exception is raised
- length can be used to check the size of an array

Arrays Example

```
public class ArrayChecker {
       // check whether an array contains nulls
       public boolean checkForNulls(Object[] array) {
             for (int i=0;i<array.length;i++) {
                    if (array[i]==null) {
                           return true;
          }
       }
       return false;
       }
}
```

loop from 0 to array.length-1

access element at position i

Arrays ctd

- because arrays are fixed length, they are not very convenient to work with
- Java contains numerous collections (lists, sets, maps) which are often better alternatives

Strings

- in Java, strings are reference types instantiating `java.lang.String`
- Java supports **string literals**: string instances can be defined by directly providing a value:
  `String name = "John";`
- + is a built-in operator to concatenate (join) string
- all other objects can be converted into a string representation using the `toString()` methods
- in many respects, string behave like arrays of characters
- the string API (= set of methods in `java.lang.String`) comprises methods to split string, extract characters, find patterns in strings, check the size of strings, etc

Mutable vs Immutable Objects

- strings are examples of **immutable objects** (can’t be changed!)
- i.e., many methods that look like they are manipulating a string are actually creating a new string

- most other classes are implemented so that their instances are mutable
- there are mutable classes to represent string in Java as well: `java.lang.StringBuilder` and `java.lang.StringBuffer`

String API Use Example

```
String s = "Hello World";
// split string - extract part up to first whitespace
String firstPart = s.substring(0,s.indexOf(' '));

System.out.println(firstPart);
```

- this will print "Hello"
- `s.indexOf(' ')` will yield 5 - the position of the whitespace character

String are Immutable

```
String s = "Hello World";
// split string - extract part up to first whitespace
String firstPart = s.substring(0,s.indexOf(' '));

System.out.println(s==firstPart);
```

String are Immutable

```
String s = "Hello World";
// split string - extract part up to first whitespace
String firstPart = s.substring(0,s.indexOf(' '));

System.out.println(s==firstPart);
```

- will print false
- `s.substring(..)` does not manipulate "Hello World", but creates a new string instead !

Comparing Strings

- because strings are so common, Java applies many optimisations to deal with strings
- this makes it risky to make assumptions about string identity
- **strings should always be compared with equals, never with == !**
- equals compares the actual characters in strings

Comparing Strings ctd

- e.g., this will return **false**:

```
String x = "b";
System.out.println("ab" == ("a"+x));
```

- however, this will return **true** as the compiler optimises this code (depending on the compiler):

```
System.out.println("ab" == ("a"+"b"));
```

- this will also return **true** as expected:

```
String x = "b";
System.out.println("ab".equals("a"+x));
```

Enums

```
public enum ColourName {
     RED,GREEN,BLUE,BLACK,WHITE,YELLOW;
}
```

- enums are special reference types
- enums enumerate a fixed set of instances of the type being defined
- these instances are accessed like static fields:

  ```
  ColourName colourName = ColourName.RED;
  ```
- enums can define methods and fields like normal classes
- enums can be used in case-switch control statements