KIT107 PROGRAMMING

Data Structures and Algorithms

Dr Julian Dermoudy
& Dr Shuxiang Xu
School of ICT

1

0. Motivation

```
211234Bloggs,John,KIT107DN,KIT102..
192435Green,Jen,BSA101CR,KIT107..
225672Kees,Bill,KIT107..,KZA101PP
208267Adams,Mandy,HSA100AN,KIT107..
.................................................
```

- What information exists in these data?
- What relationship exists between the different items?
- What operations/values are valid on each?
- Storing the information is the realm of a database manager
- Designing and modelling the data are the realm of the software engineer (i.e. us!)

KIT107 ©JRD, 2021 Slide 2

2

1. Nomenclature

- 1.1 Algorithm
- 1.2 Type
- 1.3 Data Type
- 1.4 Data Structure

KIT107 ©JRD, 2021

3

1.1 Algorithm

- An *algorithm* is a concise specification of a way to solve a problem
- Algorithms should:
  - be *finite*
  - be *deterministic*
  - be *general*
  - be *achievable*
  - be *correct*
  - have *outputs*
  - be *efficient*

KIT107 ©JRD, 2021 Slide 4

4

1.2 Type

- A *type* is a named set of all values able to be assigned to a variable of that type or referred to anonymously in an expression
- For example

```
int x;
x=32;
4+7
```

KIT107 ©JRD, 2021 Slide 5

5

1.3 Data Type

- A *data type* is a predefined, simple, unstructured type, e.g. `int`, `char`, `double`
- To use a data type we require
  - a set of simple values
  - a set of operations defined on those values

KIT107 ©JRD, 2021 Slide 6

6

1.4 Data Structure

- A *data structure* is the construct in which the information necessary for an algorithm is represented
- Usually *nouns* in a specification
- E.g. an array, which consists of:
  - finite collection of index and value pairs and a counter
  - creation, and element selection (extraction and assignment) operations

KIT107 ©JRD, 2021 Slide 7

7

2. Java

- 2.1 History
- 2.2 Types
- 2.3 Operators
- 2.4 Literal Values
- 2.5 Narrowing and Widening
- 2.6 Statements
- 2.7 Reserved Words
- 2.8 Documentation
- 2.9 Methods
- 2.10 Classes
- 2.11 Collection Framework

KIT107 ©JRD, 2021

8

2.1 History

- Java is an object-oriented programming language:
  - class hierarchy
  - class instances (*objects*)
  - objects consist of encapsulated data (*instance variables*) and behaviour (*methods*)

KIT107 ©JRD, 2021 Slide 9

9

History (Continued)

- Java consists of many pre-written classes (APIs)
  - implicitly imported from `java.lang`
  - explicitly `imported` by the programmer
- Originally called Oak, Java was released in 1995 and is based on C++ (and thus C)

KIT107 ©JRD, 2021 Slide 10

10

2.2 Types

- Java has two kinds of type:
  - *primitives* (`int`, `char`, `boolean`, `double`, `float`, `long`, `short`, `byte`)
    - variables consist of a value
  - *classes* (everything else!)
    - variables consist of a *reference* to an object
    - objects contain instance variables and methods and the member operator (`.`) is used to *dereference* (access) the object
    - *wrapper classes* exist for the primitives (`Integer`, `Character`, `Boolean`, `Double`, etc.) and *boxing/unboxing* (translation) is automatic

KIT107 ©JRD, 2021 Slide 11

11

2.3 Operators

- Logical
  - &&, ||, !
- Relational
  - ==, !=, <, <=, >, >=
- Arithmetic
  - +, –, *, /, %, +=, –=, *=, /=, %=, ++, ––
- Sequence
  - ;

KIT107 ©JRD, 2021 Slide 12

12

2.4 Literal Values

- `int`, `long`, `short`, `byte`
  - …, `-4`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `4`, …
- `double`, `float`
  - …, `-4.999`, `-4.998`, …, `-4.001`, `-4.000`, `-3.999`, …
- `boolean`
  - `true`, `false`
- `char`
  - `' '`, `'!'`, …, `'~'`
- `String`
  - `""`, `"blah"`, `etc.`
- Reference variables
  - `null`

KIT107 ©JRD, 2021 Slide 13

13

2.5 Narrowing and Widening

- Java is *strongly-typed* and won't allow arbitrary assignment of values to variables unless the types are *compatible*
- Some exceptions are implicitly allowed, others must be explicit

KIT107 ©JRD, 2021 Slide 14

14

Narrowing and Widening (Continued)

- Narrower values can be assigned to wider types, e.g.

```
int x=45;
double d;
```

✓ `d=x;`

- This is called *widening*

KIT107 ©JRD, 2021 Slide 15

15

Narrowing and Widening (Continued)

- The converse *(narrowing)* cannot be done without an explicit type cast *(coercion)*, e.g.

```
int x;
double d=17.0;

✗ x=d;
✓ x=(int)d;
```

KIT107 ©JRD, 2021 Slide 16

16

Narrowing and Widening (Continued)

- Widening and narrowing are also possible for objects, e.g.

```
public class Primate extends Animal

Primate p=new Primate();
Animal a=new Animal();

✗ p=a;
✓ a=p;
```

and now

```
✓ p=(Primate) a;
```

KIT107 ©JRD, 2021 Slide 17

17

Narrowing and Widening (Continued)

- Type casts are checked at compile-time for primitives and run-time for objects
- Coercions only change the interpretation, not the data itself
- The detection of incompatible types produces an error when the coercion is attempted, e.g.

```
✗ boolean b=(boolean)13.2;
✗ Animal a=(Animal)"String value";
```

KIT107 ©JRD, 2021 Slide 18

18

2.6 Statements

```
if (cond)
{
    block
}
else
{
    block
}
```

```
switch (expr)
{
    case val: block
              break;
    …
    default: block
}
```

KIT107 ©JRD, 2021 Slide 19

19

Statements (Continued)

```
while (cond)
{
    block
}
```

```
for (expr; cond; expr)
{
   block
}
```

KIT107 ©JRD, 2021 Slide 20

20

Statements (Continued)

```
do
{
    block
} while (cond);
```

```
try
{
    block
}
catch (Ex e)
{
    block
}
…
```

KIT107 ©JRD, 2021 Slide 21

21

2.7 Reserved Words

- Java's vocabulary include the following (which should not be used as identifiers):
  - `abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extends`, `false`, `final`, `finally`, `float`, `for`, `goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `null`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `strictfp`, `super`, `switch`, `synchronise`, `this`, `throw`, `throws`, `transient`, `true`, `try`, `void`, `volatile`, `while`

KIT107 ©JRD, 2021 Slide 22

22

2.8 Documentation

- Comments can be used to provide information to the reader
- These are ignored by the compiler
- Form:

```
// single line comment

/*
    multi line comment
*/
```

KIT107 ©JRD, 2021 Slide 23

23

Documentation (Continued)

- Class and method header comments can be automatically converted into external documentation using `javadoc`
- Tags include
  - `@author`, `@version` — classes
  - `@param`, `@return`, `@throws` — methods

KIT107 ©JRD, 2021 Slide 24

24

2.9 Methods

- A named collection of statements is called a *method*
- Information can be provided to a method (*parameters*)
  - *formal* parameters declared in the definition
  - *actual* parameters given in the method call
- A single value may be returned, or the method may be of type void
- *Constructor* methods may be declared for a class to initialise instance variables etc.

KIT107 ©JRD, 2021 Slide 25

25

Methods (Continued)

- Methods should be documented when written
  - comments in the method header
    - pre-condition — assumed to be true before
    - post-condition — guaranteed to be true after
    - informal description of purpose
    - description of parameters and return value
  - comments in the code

KIT107 ©JRD, 2021 Slide 26

26

```
/**
Method to sum given number of array elements
@param numbers array to be summed
@param length index to sum up to (exclusive)
@return int sum of specified elements
Precondition: array instantiated, length greater than 0
Postcondition: all elements summed up to length, result
               returned
*/
public int sum(int []numbers, int length)
{
   int ans; // sub-total of summation

   // sum the array elements
   for (int i=0; i<length; i++)
   {
         ans+=numbers[i];
   }

   return ans; // final result
}
```

27

Methods (Continued)

- Pre-conditions can be enforced at run-time through *assertions*
  - Failure of an assertion results in an `AssertionError`
- Assertions are usually placed at the start of methods (after local variable declarations)
- Example (for program on previous slide)
  - `assert (length>0);`

KIT107 ©JRD, 2021 Slide 28

28

Methods (Continued)

- Methods can also be recursive
- A recursive method is one which is defined in terms of itself
- There needs to be
  - a part of the method which ends the method and
  - a part of the method which simplifies the problem and repeats

KIT107 ©JRD, 2021 Slide 29

29

```
/**
Method to sum given number of array elements
@param numbers array to be summed
@param length index to sum up to (exclusive)
@return int sum of specified elements
Precondition: array instantiated, length greater than 0
Postcondition: all elements summed up to length, result
               returned
*/
public int sum(int []numbers, int length)
{
   if (length == 0) // no numbers so answer is zero
   {
         return 0;
   }
   else
   {
         // sum the array elements
         return numbers[length+1] +
                     sum(numbers,length-1);
   }
}
```

30

2.10 Classes

- A class is a concept/blue-print/mould/factory/etc
- It defines the knowledge/state of objects of its type (instance variables), and their behaviour/abilities (methods)
- Every instance of a class gets its own instance variables and methods

KIT107 ©JRD, 2021 Slide 31

31

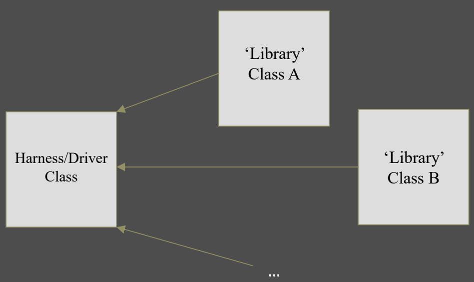
Classes (Continued)

- Java applications:
  - not event driven (unless a GUI present)
  - entry point is `main()` method
  - no GUI (by default)
  - input using `java.util.Scanner` methods
  - output using `System.out.print()` and `System.out.println()`
  - compiled with DrJava or `javac`

KIT107 ©JRD, 2021 Slide 32

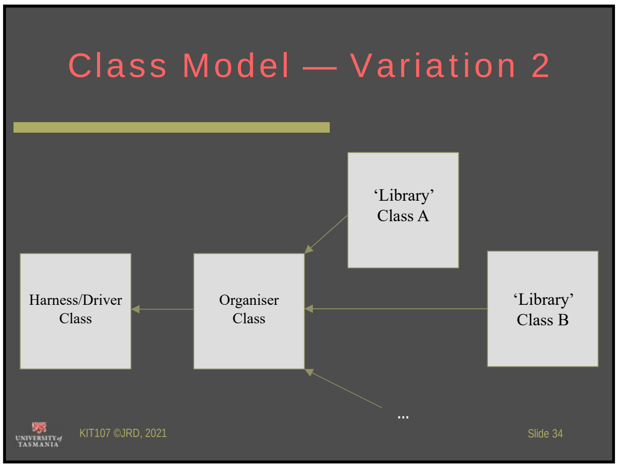
32

Class Model — Variation 1



KIT107 ©JRD, 2021 Slide 33

33



34

Harness Class

```
import java.util.Scanner;
import LibraryClass;

public class ExampleHarness
{
    public static void main(String args[])
    {
            local variable declarations

            statements
    }
}
```

KIT107 ©JRD, 2021 Slide 35

35

Library/Organiser Class

```
public class Example
{
    instance variable declarations

    public Example()
    {
            local variable declarations

            statements
    }

    other user-defined methods
}
```

36

Library/Organiser Class (Continued)

- Good practice:
  - `private`/`protected` instance variables
  - `public` methods
    - *constructor(s)* (and possibly *deconstructor(s)*)
    - *getter(s)* for each instance variable
    - *setter(s)* for each instance variable
    - *doer(s)* for activities of the object

KIT107 ©JRD, 2021 Slide 37

37

2.11 Collection Framework

- Java contains a framework of *collection* abstract data types and other data structures including
  - `Array`, `ArrayList`, `Arrays`, `LinkedList`, `List`, `Map`, `PriorityQueue`, `Queue`, `Set`, `SortedMap`, `SortedSet`, `Stack`, `TreeSet`, and `Vector`
- You are *not* to use them in this unit (or you won't learn anything!)

KIT107 ©JRD, 2021 Slide 38

38

3. Addresses, References, and Dynamic Variables

- 3.1 Addresses
- 3.2 Parameter Passing — Call-by-Value

KIT107 ©JRD, 2021

39

3.1 Addresses

- Memory consists of locations
- Each location is labelled with an *address*
- Addresses are usually hexadecimal (base 16) integers

KIT107 ©JRD, 2021 Slide 40

40

```
public class Example
{
     public static void main(String args[])
     {
          int x;
          x=30;
          System.out.println("x is " + x);
     }
}
```

Memory

1	30	(int)x
2		

KIT107 ©JRD, 2021 Slide 41

41

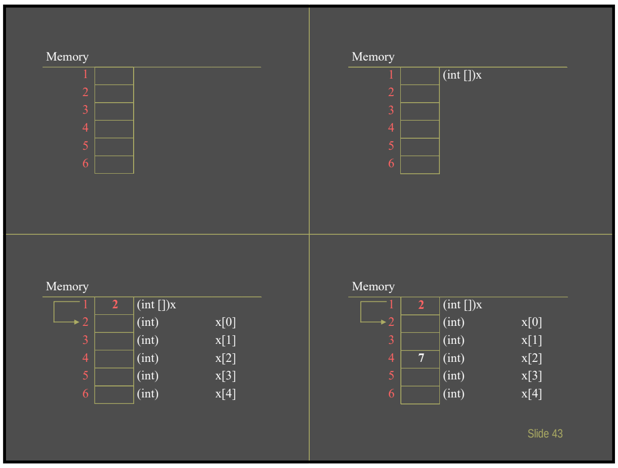
```
public class Example
{
     public static void main(String args[])
     {
          int x[];
          x=new int[5];
          x[2]=7;
          System.out.println("x[2] is " + x[2]);
     }
}
```

Memory

1	2	(int [])x	
2		(int)	x[0]
3		(int)	x[1]
4	7	(int)	x[2]
5		(int)	x[3]
6		(int)	x[4]

KIT107 ©JRD, 2021 Slide 42

42



43

Dynamic Variables

- `int x[];` space is reserved only for the variable x, not for the elements of the array
- Similarly for object variables, e.g. variables of type `Button`, `String`, `TextField`, `Label`, `Integer`, `Character`, and so on
- Such variables are called *dynamic variables*

KIT107 ©JRD, 2021 Slide 44

44

References

- `x = new int[10];` space is dynamically reserved for the elements of the array and the variable x is given the address of the first of these as its value
- the address stored in the variable x is called a *reference*
- the variable x is called a *reference variable*

KIT107 ©JRD, 2021 Slide 45

45

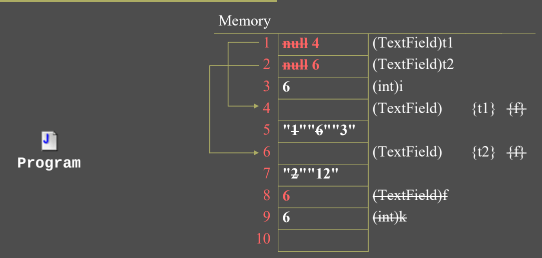
3.2 Parameter Passing

- All parameter passing is call-by-value
  - the type and number of parameters is checked
  - the value of the actual parameter is copied into a newly created variable (the formal parameter)
  - the method is executed
  - local variables (including formal parameters) of the called method are deleted

KIT107 ©JRD, 2021 Slide 46

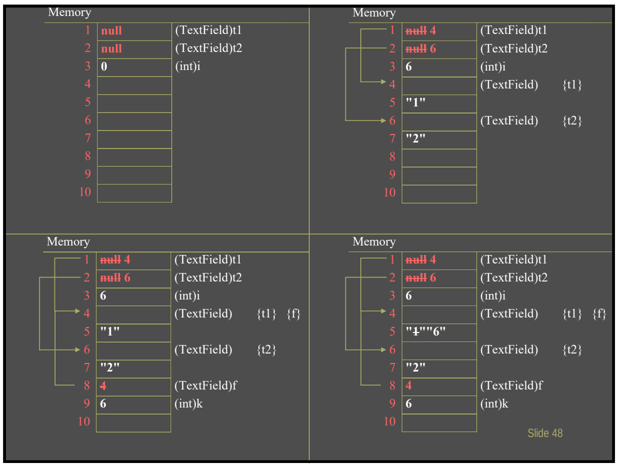
46

Example

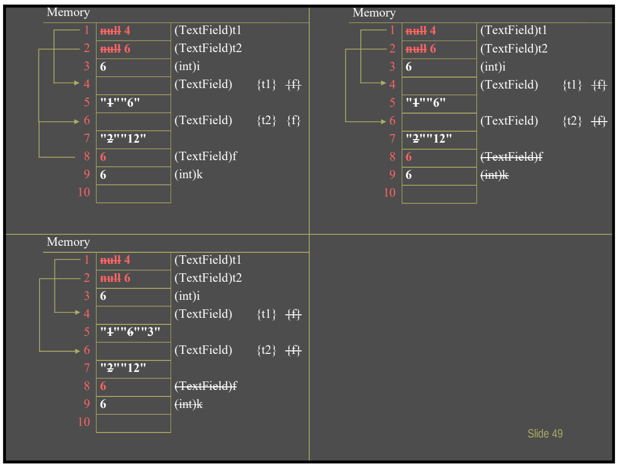


KIT107 ©JRD, 2021 Slide 47

47



48



49

4. Abstract Data Types

- 4.1 Abstract Data Types
- 4.2 Representation
- 4.3 Example: Modelling a Time
- 4.4 ADTs in Java
- 4.5 Access Modifiers

Additional References

KIT107 ©JRD, 2021

50

4.1 Abstract Data Types

- Program language independent concept
- Describe the structure of the data being manipulated
- Capture the relationships between different components of the data
- Encapsulates the operations available on the data with the data

KIT107 ©JRD, 2021 Slide 51

51

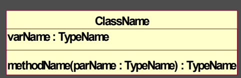
4.2 Representation

- ADTs can be modelled in many forms:
  - individual-specific formats
  - language-specific formats, e.g. Java and C
  - ADT signatures (a formal, language-independent, text-based mechanism)
  - class diagrams (in a standardised format such as UML)

KIT107 ©JRD, 2021 Slide 52

52

Unified Modelling Language

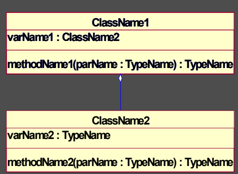


- UML is diagramatical
- Classes are represented as rectangles with three sections:
  - name
  - instance variables
  - methods

KIT107 ©JRD, 2021 Slide 53

53

Unified Modelling Language

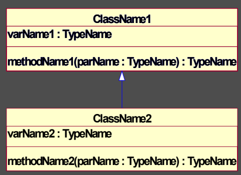


- ‘Has-a’ relationships (aggregation) exists when one class contains a variable of the type of another class

KIT107 ©JRD, 2021 Slide 54

54

Unified Modelling Language



- 'Is-a' relationships (specialisation/ generalisation) exists when one class is the subclass of another class

KIT107 ©JRD, 2021 Slide 55

55

4.3 Example: Modelling a Time

- Pick a time of day
- What does it consist of?
  - an hour, a minute, and a second
- What can you do with a time value?
  - change it, share it (in AM/PM format and in 24hr format), and compare it with another time value
- How can you model the concept and implement it?

KIT107 ©JRD, 2021 Slide 56

56

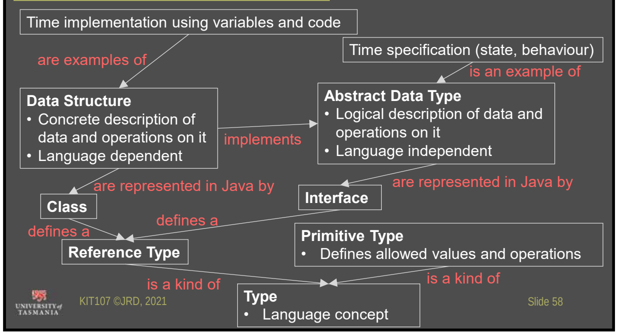
Two levels

- Conceptual
  - Abstract Data Type Specification
- Concrete
  - Implementation using data structures

KIT107 ©JRD, 2021 Slide 57

57

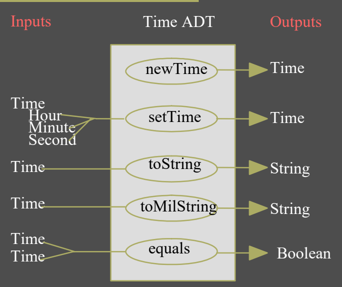
Terminology Relationships



KIT107 ©JRD, 2021 Slide 58

58

Example: Time (black box diagram)



KIT107 ©JRD, 2021 Slide 59

59

Example: Time (UML diagram)

Time
hour : Hour minute : Minute second : Second
Time() : Time setTime(hour : Hour, minute : Minute, second : Second) : void toString() : String toMilitaryString() : String equals(that : Time) : Boolean

KIT107 ©JRD, 2021 Slide 60

60

UML Diagrams

- UML diagrams specify:
  - the definition of a new type
  - the components/structure of the new type
  - the *types* required for the implementation of the new type
  - the form of the available operations on variables of that type

KIT107 ©JRD, 2021 Slide 61

61

Aside 1: Where did the Time go?

- Q: The black box diagram showed the operations (`setTime()`, `toString()`, `toMilitaryString()`, and `equals()`) required a `Time` value as a parameter — where did it go?
- A: UML and Java are object-oriented — all the operations are inside a `Time` variable already (the `this` object)!

KIT107 ©JRD, 2021 Slide 62

62

Aside 2: ADTs vs Data Types and Data Structures

- Q: Why use the type “Hour”, “Minute”, and “Second” — why not use “`int`”?
- A:
  - the concepts are different
    - hours aren’t minutes, minutes aren’t seconds
  - the values are different
    - 0<=hour<=12, 0<=minute,second<=59
  - the operations are different
    - some `int` operations, e.g. % cannot be performed on Hours, Minutes, or Seconds values

KIT107 ©JRD, 2021 Slide 63

63

4.4 ADTs in Java

- A Java mechanism(s) is required that:
  - allows data and operations to be declared
  - enforces security
  - facilitates portability and code re-use
  - separates specification from implementation
  - encapsulates data and operations

64

Java Interface

- The Java mechanism for the ADT specification is the interface
- method *headings* are included (other than the constructor) but nothing else

```
public interface TimeInterface
{
        //public Time();
        public void setTime(Hour h, Minute m, Second s);
        public String toString();
        public String toMilitaryString();
        public boolean equals(Time t);
}
```

65

Java Interfaces and the Compiler

- All methods declared in an interface must be defined in a class that *implements* the interface
- Error messages (“class must be declared abstract”) will ensue if this is not completed
- All classes mentioned must exist — sometimes it is expedient to use an existing data type/ structure rather than create more classes

66

Java Implementation

Implementation

67

Harness (Client) Classes

- ADT implementations are 'passive' (library classes)
- A client/harness class is required in order for the ADT to be used to achieve some task
- The same ADT can be used for many differing purposes, the client/harness class is different for each purpose

68

Example Harness

```
public class ExampleHarness
{
    public static void main(String args[])
    {
        Time t;

        t = new Time();
        System.out.println("The time is " + t.toString());
    }
}
```

Project

69

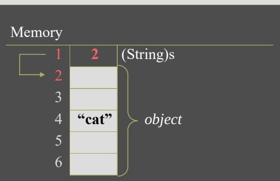
Instantiation

- When the new keyword is used memory is reserved for an *object* of that class
- Once the memory is reserved the *constructor* is executed
- This is the process of *instantiation*

KIT107 ©JRD, 2021 Slide 70

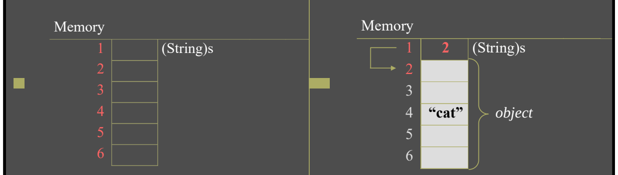
70

```
public class Example
{
     public static void main(String args[])
     {
          String s;
          s=new String("cat");
          System.out.println("string is " + s);
     }
}
```



KIT107 ©JRD, 2021 Slide 71

71



KIT107 ©JRD, 2021 Slide 72

72

```
public class Example
{
      public static void main(String args[])
      {
            String s;
            System.out.println("length is " + s.length());
      }
}
```

Project

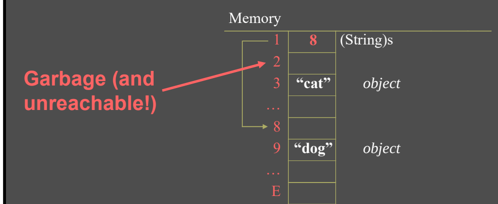
73

Garbage Collection

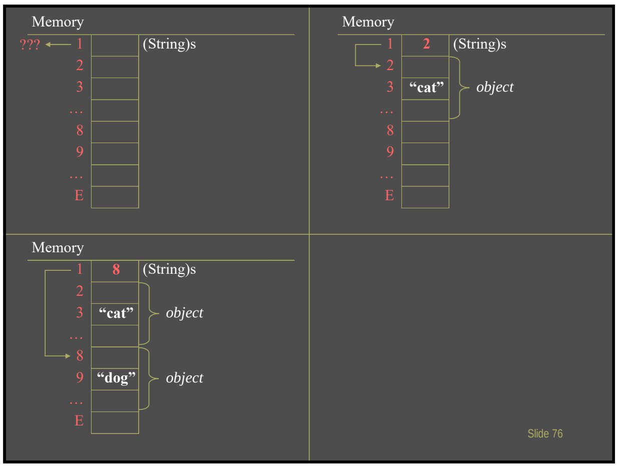
- Objects can become unreachable when their reference variable(s) is/are assigned to a different address
- When this happens the object is *garbage*
- Java runtime systems possess a *garbage collector* to free memory

74

```
public class Example
{
      public static void main(String args[])
      {
            String s;
            s=new String("cat");
            s=new String("dog");
      }
}
```



75



76

Overloading

- Multiple method definitions may be defined with the same name but differing parameters (*overloaded*)

```
public Time()
{
        setTime(0,0,0);
}

public Time(int h)
{
        setTime(h,0,0);
}
```

KIT107 ©JRD, 2021 Slide 77

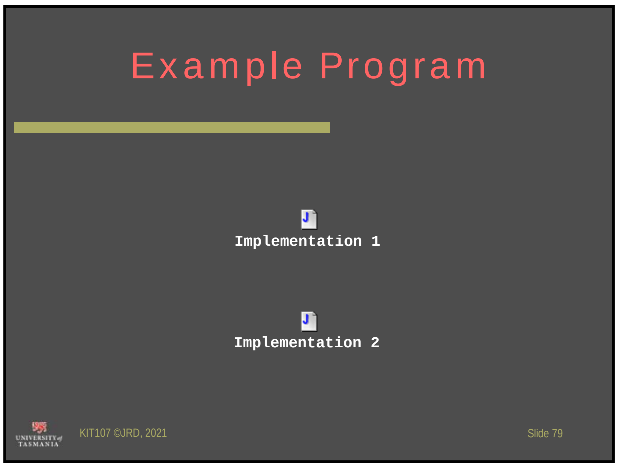
77

4.5 Access Modifiers

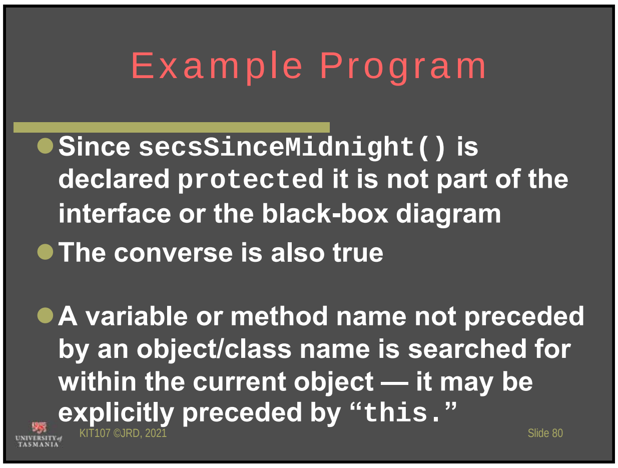
- `public`
- `friend`
- `protected`
- `private`

KIT107 ©JRD, 2021 Slide 78

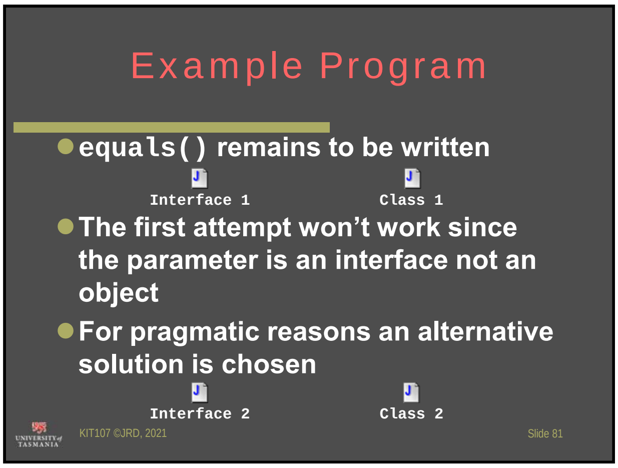
78


Example Program
Implementation 1
Implementation 2
KIT107 ©JRD, 2021
Slide 79

79


Example Program
Since secsSinceMidnight() is declared protected it is not part of the interface or the black-box diagram
The converse is also true
A variable or method name not preceded by an object/class name is searched for within the current object — it may be explicitly preceded by "this."
KIT107 ©JRD, 2021
Slide 80

80


Example Program
equals() remains to be written
Interface 1
Class 1
The first attempt won't work since the parameter is an interface not an object
For pragmatic reasons an alternative solution is chosen
Interface 2
Class 2
KIT107 ©JRD, 2021
Slide 81

81

5. The Stack ADT

- 5.1 The Stack ADT
- 5.2 Polymorphism and Genericity
- 5.3 Syntax vs Semantics
- 5.4 User-Defined Exceptions
- 5.5 Primitive Operations vs Derived Operations
- 5.6 Stack Implementation in Java (Using Arrays)

KIT107 ©JRD, 2021

82

5.1 The Stack ADT

- A *Stack* is either empty (*null*) or consists of a first element (the *top-of-stack*) and the remainder of the stack which is itself a stack
- The Stack is a *recursive* (or *self-referential*) data structure

KIT107 ©JRD, 2021 Slide 83

83

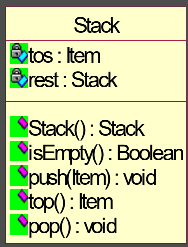
Stack Structure and Operations

- Last-In-First-Out structure
- Only *top of stack* visible
- Can only *push* new items onto top
- Can only *pop* items off the top
- Example: stack of plates, clothes on the floor, post-fix calculator

KIT107 ©JRD, 2021 Slide 84

84

The Stack ADT Diagram



KIT107 ©JRD, 2021 Slide 85

85

The Stack ADT Java Interface

```
public interface StackInterface
{
   //public Stack();
   public boolean isEmpty();
   public void push(Item i);
   public Item top();
   public void pop();
}
```

KIT107 ©JRD, 2021 Slide 86

86

5.2 Polymorphism and Genericity

- **How can any kind of item be represented?**
- **Java has a class called "`Object`"**
- **`Object` is the *base class* of all Java classes (i.e. all classes have `Object` as an ancestor class and `Object` has no parent class)**

KIT107 ©JRD, 2021 Slide 87

87

Polymorphism and Genericity (continued)

- Any object is of its declared type, the type of its superclass, and of all its ancestor classes (including Object)
- Each object is then of multiple types simultaneously — it is *polymorphic*

KIT107 ©JRD, 2021 Slide 88

88

Polymorphism and Genericity (continued)

- Thus a Stack of `Objects` can be a stack of anything — the stack is *generic*
- (This brings two problems that we will solve later: consistency of type content and necessity for type casting)

KIT107 ©JRD, 2021 Slide 89

89

The Stack ADT Java Interface

```
public interface StackInterface
{
   //public Stack();
   public boolean isEmpty();
   public void push(Object o);
   public Object top();
   public void pop();
}
```

KIT107 ©JRD, 2021 Slide 90

90

5.3 Syntax vs Semantics

- UML diagrams and Java interfaces only indicate the *syntax* (form)
- Neither indicate the *semantics* (meaning)
- *Axioms* could provide the semantics

```
For all values i of Item, and values s of Stack:
    isEmpty(newStack()) = true
    isEmpty(push(i,s)) = false
    top(push(i,s)) = i
    top(newStack()) = ERROR
    pop(push(i,s)) = s
    pop(newStack()) = ERROR
```

91

The Stack ADT Java Interface

```
public interface StackInterface
{
   //public Stack();
   public boolean isEmpty();
   public void push(Object o);
   public Object top() throws
               EmptyStackException;
   public void pop() throws
               EmptyStackException;
}
```

KIT107 ©JRD, 2021 Slide 92

92

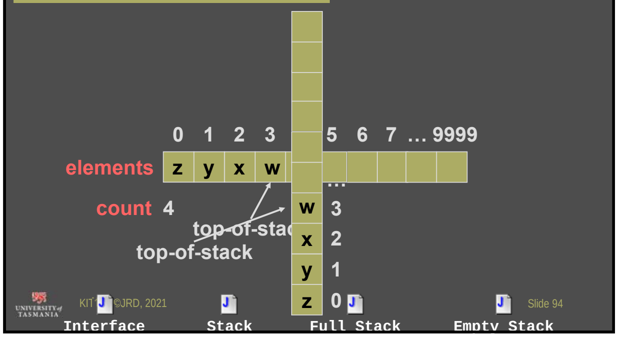
5.4 User-Defined Exceptions

- Code that may give rise to an exception is attempted by prefacing it with `try`
- When an exception occurs, control is passed to the `catch`
- Exceptions may be raised deliberately using `throw`
- Methods can indicate exceptions could occur using `throws` clauses

KIT107 ©JRD, 2021 Slide 93

93

5.6 Stack Implementation using Arrays



KIT107 ©JRD, 2021 Slide 94

94

C

- What would all of this look like in C (which is a *procedural* programming language)?

KIT107 ©JRD, 2021 Slide 95

95

Object-Oriented Programming

- Classes encapsulate state and behaviour of objects
- One namespace per class
- Objects are instances of the class
- Methods are invoked on the object

```
public class X
{
     protected int y;

     public void m()
     {
          …
     }
}

…
X x = new X();✓
x.m();        ✓
x.y = 38;     ✗
```

KIT107 ©JRD, 2021 Slide 96

96

Procedural Programming

- Variables and functions are not contained in anything and hence not related to each other
- One namespace for all files (not each file)
- No instantiation
- Variables are passed to functions (procedures)

```
int y;

void m(int a)
{
     …
}

…
m(y);      ✓
y = 38;    ✓
```

KIT107 ©JRD, 2021

Slide 97

97