FIT2099 Notes

1. Table of Contents

2. Good Design in Software

Some Combination of - Functionaly correct - Performs well enough - Usable - Reliable - Maintainable

these are the properties of the system, not any design artifacts

2.1. Dependencies

2.1.1. Dependency Control

- Biggest issue in design
- Controlling the extent of dependencies

- Controlling the nature of dependencies

Will have some depenencies, having fewer dependencies makes it easier to debug, modify, change the component

- form of depencies matter

2.1.2. Why Dependencies

- dependencies are unavoidable
- if code unit A depends on code unit B
- Bugs in B may manifest in A
- Changes to B may require changes to A
- **Dependencies have to:**
- *only present when necessary*
- *explicit*
- *easy to understand*

2.2. Connascence

- Based on earlier ideas of *cohesion* and *coupling*

```
two components are connascent if a change in one would require the other to be modified in o
            - Connascence (Wikipedia Definition)
```

2.2.1. Importance of Connascence

More connascence means: - Harder to extend. - More chance of bugs - Slower to write in the first place

- Not all instances are equal!
- In general, later-listed ones are worse than others.
- Locality matters!
  - Within a method -> almost (but not totally) irrelevant.
  - Between two methods in a class -> often no big deal.
  - Two classes -> warning warning
  - Two classes in different packages -> WARNING WARNING
  - Across application boundaries -> keep to absolute minimum.
- Explicitness matters

2.2.2. Type of Connascence

Type	Description	Example
Static	obvious from code structure, auto identified by IDE	
Dynamic	Dynamically Generated	

2.2.2.1. Connascence of Name Type: **Static** has no argument(s)

```
class Watch {
    public void testWatch(){
        ...
    }
}
```

called using

```
class Hello {
    Watch demo = new Watch();
    watch.testWatch();
}
```

2.2.2.2. Connascence of Type Type: **Static**

has argument

```
class Watch {
    public void testWatch(int maxTick){
        ...
    }
}
```

called using:

```
class Hello {
    Watch demo = new Watch();
    watch.testWatch(1000);
}
```

2.2.2.3. Connascence of Position Type: **Static** - where order of which things go

```
public LinkedCounter(LinkedCounter l, Counter neighbour){
    super(l);
    this.neightbour = neighbour;
}
```

watch 3:

```
public Watch3(Watch3 w){
    this.hours = new MaxCounter(w. hours);
    this.minutes = new LinkedCounter (w.minutes, this.hour);
    this.seconds = new LinkedCounter (w.seconds, this.minutes);
}
```

It has to remember the position for example `this.minutes = new LinkedCounter (w.minutes, this.hour);` has to remember the position of this.hour

2.2.2.4. Connascence of Meaning/Convention (CoM/CoC) Type: **Static**

```
public void increment(){
    super.increment();
    if(this.getValue() == 0){
        neighbour.increment();
    }
}

public void reset(){
    value = 0;
}
```

- Documentation is **important**

2.2.2.5. Connascence of Algorithm Type: **Static**

```
1. (message, key) ->  Encrypter
2. Encrypted Messages trasmits
3. Encrypted Message Must implement reverse of encrypter
```

must document very precisely

IPoAC - https://en.wikipedia.org/wiki/IP_over_Avian_Carriers

2.2.2.6. Connascence of Execution (CoE) Type: **Dynamic**

Example:

```
public Watch3(){
    hours = maxCounter(24):
    minutes = new LinkedCounter(60, hours);
    seconds = new LinkedCounter(60, minutes);
}
```

Must be ran in the right order for example, hours must be run first (variable declaration)

2.2.2.7. Connascence of Timing (CoT) Type: **Dynamic**

- Parllel Computing
- Interacting with hardware - especially real-time computing
- Distributed Computing

2.2.2.8. ## Apollo 11 Example

- Requested available memory
- Other programs
- Constant Reboot

2.2.2.9. Connascnce of Values (CoV) Type: **Dynamic**

Where two values (variables) must be equal (the same) and if changes, it has to be changed as well

2.2.2.10. Connascnce of Identity (CoI) Type: **Dynamic**

When two or more variables has to point the object

2.3. Contrascence

- When two things are required to be different
- This is a form of connascence
- "Aliasing bugs" – an example fault type where contranescence has not been maintained.

2.4. Minimising Connascence

1. Minimise overall amount of connascence by breaking system into encapsulated elements.
2. Minimise remaining connascence that crosses encapsulation boundaries (guideline 3 will help with this)
3. Maximise connascence within ecapsulation boundaries

3. Encpsulation

3.1. What is Encapsulation?

- a software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module
- the concept that access to the names, meanings, and values of the responsibilities of a class is entirely separated from access to their realization.
- the idea that a module has an outside that is distinct from its inside, that it has an external interface and an internal implementation

3.2. Mechanisms

- Java was made to encapsulate.
- Basic unit of Java programs is the class.
- Can restrict access to anything in the class to:
  - Within the class only (`private`)
  - Within the package only (no access modifier - default)
  - Only to subclasses and within the package (`protected`)
  - No restrictions (`public`)

3.3. Using Encapsulation in Java

- Avoid public attributes
- Only make methods public where necessary.
- Keep the class package-private if not needed!
- Use `protected` sparingly, consider using methods rather than attributes

3.4. Defensively Copy

- When getters return a mutable object.

- One with public attributes or mutator methods other than constructor.
- Generally, make a copy and return that.
- Otherwise, lose benefit of encapsulation and control of connascence...

4. Designing Software

4.1. Starting

- Start at the Top:
  - Start with high-level problem.
  - Divide into subproblems.
  - Design to solve those.
  - Put it together...
- Start at the bottom:
  - Start with a small problem that you can solve.
  - Design a solution to that.
  - Do a few more...
  - Start putting them together
  - Voila...a solution!

4.2. Scenario based design

- Have some scenario(s) that the thing being designed needs to support.
  - Storyboard, activity diagram, plain text...
  - This may come out of requirements or analysis (depending on whether thing is "the system" or some small part of it)
- Work through your scenario(s).
  - Trace through your design as it stands.
- Modify/rework design to support scenario effectively.
  - Keep quality properties in mind...
- Repeat with additional scenarios.

5. Using Abstraction in Java (Week 8 Lecture 2)

5.1. Using Abstraction at Code Level

- Abtraction is a **design principle** rather than a *programming technique*
- You do not have to write generic classes in this unit

5.2. Features of Java

5.2.1. Class

- is the most important mechnaism in most OO Languages (incl. Java)
  - represent single concept
  - expose a public interface that allows response in order to furfill its responsiblity
  - hide any implementation details that don't directly fullfil that responsiblity
  - ensures that its attribution are in a valid condition rather than relying on client code to maintain its state

5.2.2. Visibiltiy Modifiers

- These include `public`, `private` and `protected`
- in general when in doubt make it `private`
- only provide `getters` and `setters` if you're sure that external classes need to directly manipulate
- if you leave the visibility modifier, your class/attribute/method will be visible within the package which is declared.

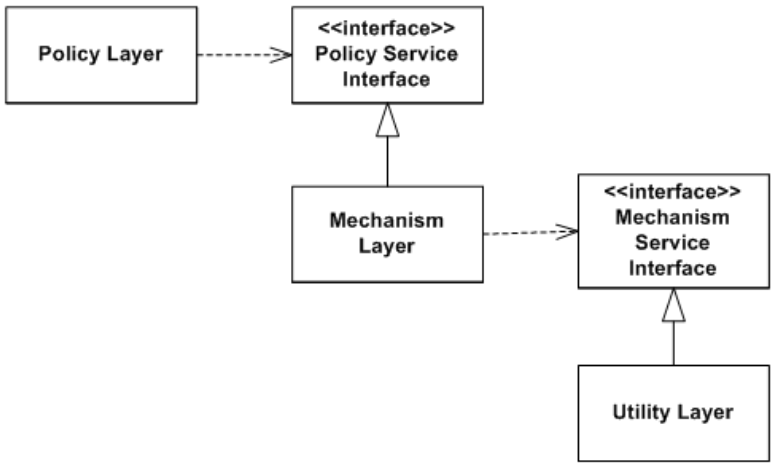
5.2.3. The Abstract Class

- The `abstract` class cant be instantiated
  - may lack important components
  - such as method bodies
  - inherits the methods and attributes, this means that it can implement the public methods and the attributes specified by the **base** class.

5.2.4. Hinge Points

- Applying dependency inversion to a single relationship.

- We take a class, seperately define it's interface as an abstract entity, seperate the code. We can let the client code interact with the abstract interface. They only interact with each other through the interface.



5.2.5. Packages

We want to split things up into packages. - We group a bunch of `classes` and bundle it into a subsystem. - The boundary around a package is also an encapsulation boundary.

5.2.6. Nesting Packages

- You can't put a package inside another package in Java
- `java.util.jar` is not a package within `java.util`
- If you want to use the package, you have explicitly import (e.g. `import java.util`)

5.2.7. Abtraction Layers

- An **abstraction layer** is the publicly accessible interface to a class, package or subsystem.
- You can create an abstraction layer by restricting visiblity as much as possible.
- One problem is to making too much public.

6. FIT2099 Week 9 Lecture A

Student data type

```
Name
StudentID
Address
```

- charactersitics behind system
- System support. Given a `studentID` return `studentName`
- Find specification of the class.

6.1. Client Supplier Relationship

- We can draw the UML

Client -> Supplier

- **Client** Watch1 "has" 2 counter attributes.
- Client is a supplier of services to `Watch1`
- `Watch1` is a client of Counter, and asks it to perform services such as `increment`, `reset()`
- Inheritments making use of service to.

6.2. Software Spec: The Problem

- Hardware components
- Well-edfined public interafcaes with a hidden implementation
- Have regorous umabgiousous *specification* of behaviour

6.3. Design by Contract

- *class* desginer establishes a *software contract* between him/herselfs and the user(s) of the class he/she designs
- make this impersonal. Contract betweent he class that is the supplier and the clients of the class

6.4. Software Contract

- Documentation of the class of the technical user
- the possiblity of enforcing the contract by using exceptions and assertions

Software Contract:

```
Class Documentation

public class Documentation{

}
```

- Software designer tells the user what the class does by providing specs for the class
- What the methods of the class need to operate correctly e.g. `assert studentID to be interger and between 00000001 to 99999999`
- What the class will guarantee to be if is used correctly

6.5. Specification of a Class

- A specification
  - is ideally part of the implementation
    - In some languages such as *Eiffel* that is built ibn others that i can done by hand (via the use of assertions and exceptions)
  - There are also extendetions
    - Cofoja (Java)
    - Py Contracts (python)
    - Spec## and Code Contract from Microsoft Research for C## and .Net
  - Should ideally be extractable from the implementation via a tool.
    - e.g. Javadoc when using Cofoja
  - is esetnail supporting component reuse and maintenance
  - is more that just the API we havve gotten used to seeing
    - it includes **comments**, and crucially exexcutable sepcs
- The User:
  - should be able to derermine how to use the class
  - not have to look at implemenation details
- Specs forms the public interface of the class

6.6. Specs

- Preconditions ('requires')
  - things that need to be true for method to run

7. UML Diagram

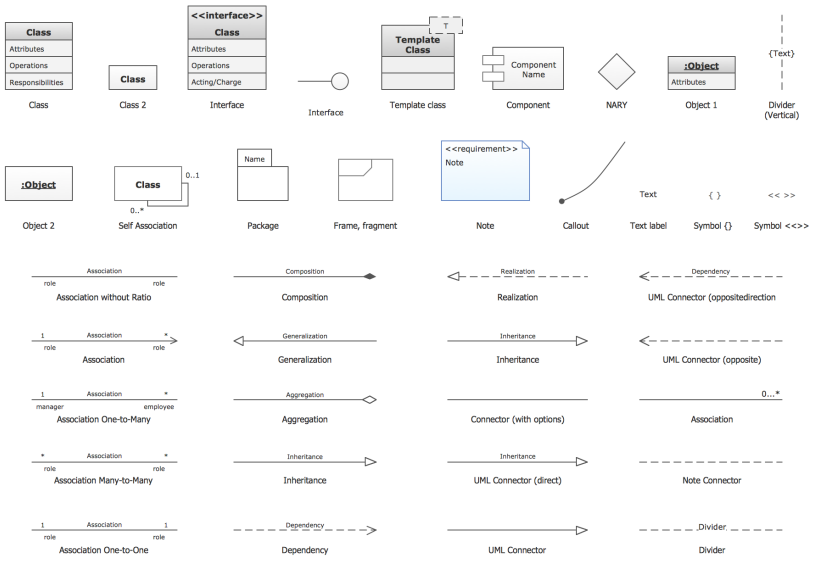


Figure 1: img

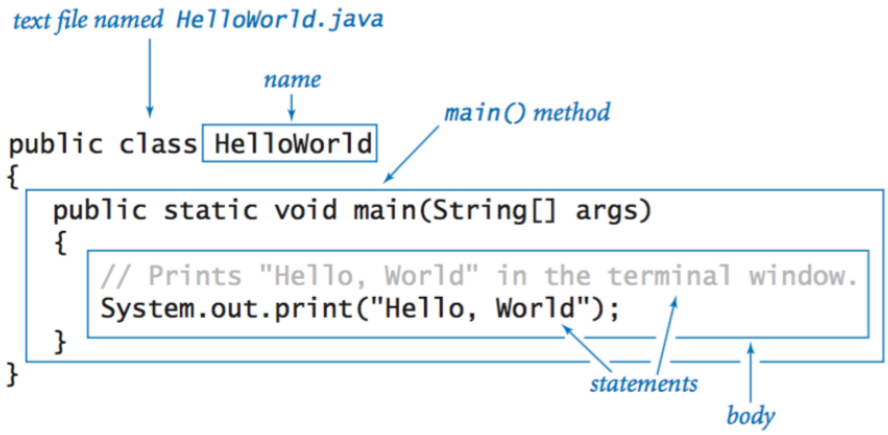


Figure 2: img

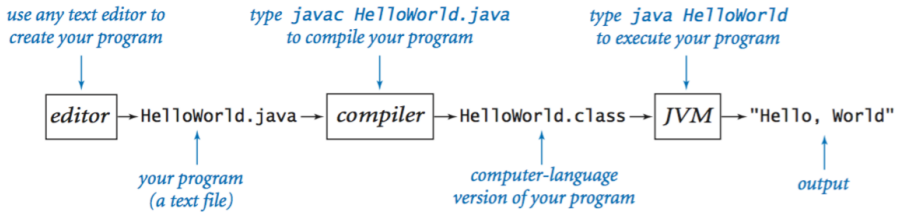


Figure 3: img2

type	set of values	common operators	sample literal values
int	integers	+ - * / %	99 12 2147483647
double	floating-point numbers	+ - * /	3.14 2.5 6.022e23
boolean	boolean values	&& || !	true false
char	characters		'A' '1' '%' '\n'
String	sequences of characters	+	"AB" "Hello" "2.5"

Figure 4: img3

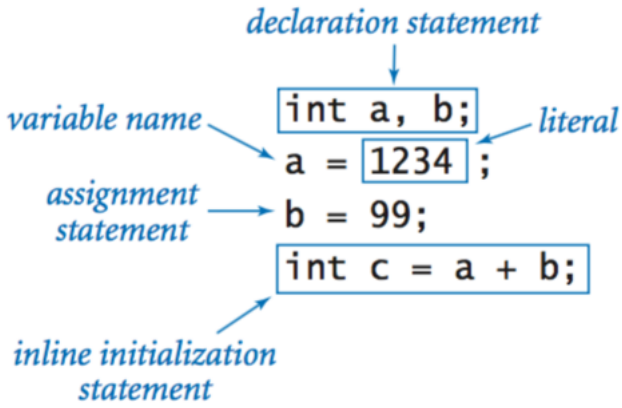


Figure 5: img4

8. Java Cheat Sheet

Classes

Editing, compiling and executing

Built-in data types

Declaration and Assignment Statements

Integers

values	integers between -2^{31} and $+2^{31}-1$					
typical literals	1234 99 0 1000000					
operations	*sign*	*add*	*subtract*	*multiply*	*divide*	*remainder*
operators	+ -	+	-	*	/	%

Figure 6: img5

expression	*value*	*comment*
`99`	99	*integer literal*
`+99`	99	*positive sign*
`-99`	-99	*negative sign*
`5 + 3`	8	*addition*
`5 - 3`	2	*subtraction*
`5 * 3`	15	*multiplication*
`5 / 3`	1	*no fractional part*
`5 % 3`	2	*remainder*
`1 / 0`		*run-time error*
`3 * 5 - 2`	13	** has precedence*
`3 + 5 / 2`	5	*/ has precedence*
`3 - 5 - 2`	-4	*left associative*
`( 3 - 5 ) - 2`	-4	*better style*
`3 - ( 5 - 2 )`	0	*unambiguous*

Figure 7: img6

values	real numbers (specified by IEEE 754 standard)			
typical literals	`3.14159`	`6.022e23`	`2.0`	`1.4142135623730951`
operations	*add*	*subtract*	*multiply*	*divide*
operators	`+`	`-`	`*`	`/`

Figure 8: img7

expression	*value*
`3.141 + 2.0`	`5.141`
`3.141 - 2.0`	`1.111`
`3.141 / 2.0`	`1.5705`
`5.0 / 3.0`	`1.6666666666666667`
`10.0 % 3.141`	`0.577`
`1.0 / 0.0`	`Infinity`
`Math.sqrt(2.0)`	`1.4142135623730951`
`Math.sqrt(-1.0)`	`NaN`

Figure 9: img8

method call	*library*	*return type*	*value*
`Integer.parseInt("123")`	`Integer`	`int`	`123`
`Double.parseDouble("1.5")`	`Double`	`double`	`1.5`
`Math.sqrt(5.0*5.0 - 4.0*4.0)`	`Math`	`double`	`3.0`
`Math.log(Math.E)`	`Math`	`double`	`1.0`
`Math.random()`	`Math`	`double`	*random in* $[0, 1)$
`Math.round(3.14159)`	`Math`	`long`	`3`
`Math.max(1.0, 9.0)`	`Math`	`double`	`9.0`

Figure 10: img10

expression	*expression type*	*expression value*
`(1 + 2 + 3 + 4) / 4.0`	`double`	`2.5`
`Math.sqrt(4)`	`double`	`2.0`
`"1234" + 99`	`String`	`"123499"`
`11 * 0.25`	`double`	`2.75`
`(int) 11 * 0.25`	`double`	`2.75`
`11 * (int) 0.25`	`int`	`0`
`(int) (11 * 0.25)`	`int`	`2`
`(int) 2.71828`	`int`	`2`
`Math.round(2.71828)`	`long`	`3`
`(int) Math.round(2.71828)`	`int`	`3`
`Integer.parseInt("1234")`	`int`	`1234`

Figure 11: img9

Floating-point numbers

Type Conversion and Library Calls

If and else

absolute value	`if (x < 0) x = -x;`

put the smaller value in x and the larger value in y	`if (x > y)`
	`{`
	`  int t = x;`
	`  x = y;`
	`  y = t;`
	`}`

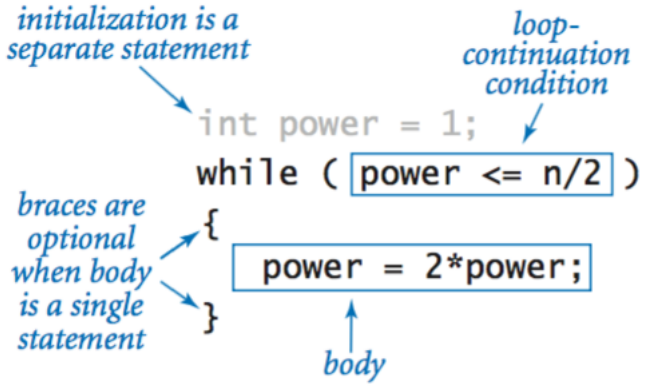
maximum of x and y	`if (x > y) max = x;` `else  max = y;`
error check for division operation	`if (den == 0) System.out.println("Division by zero");` `else  System.out.println("Quotient = " + num/den);`

error check for quadratic formula	`double discriminant = b*b - 4.0*c;`
	`if (discriminant < 0.0)`
	`{`
	`  System.out.println("No real roots");`
	`}`
	`else`
	`{`
	`  System.out.println((-b + Math.sqrt(discriminant))/2.0);`
	`  System.out.println((-b - Math.sqrt(discriminant))/2.0);`
	`}`

Figure 12: if-else

Switch Statement

```
switch (day) {
   case 0: System.out.println("Sun"); break;
   case 1: System.out.println("Mon"); break;
   case 2: System.out.println("Tue"); break;
   case 3: System.out.println("Wed"); break;
   case 4: System.out.println("Thu"); break;
   case 5: System.out.println("Fri"); break;
   case 6: System.out.println("Sat"); break;
}
```



Loops

Functions

Constructors

Instance Methods

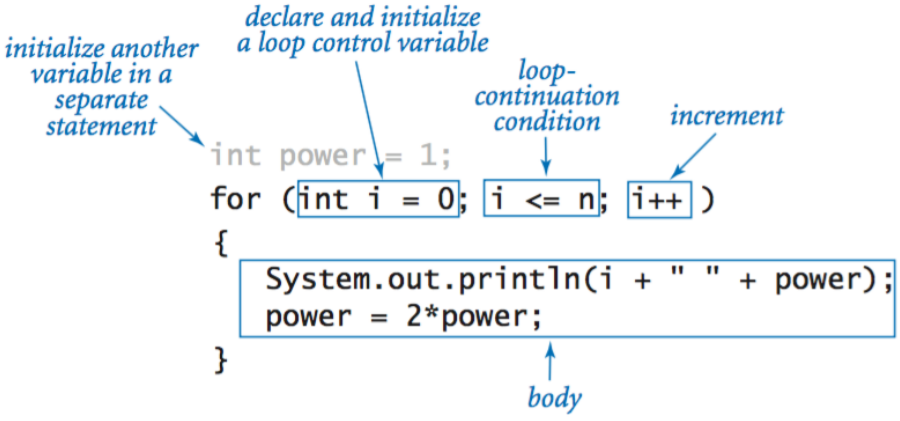


Figure 13: for

absolute value of an `int` value

```
public static int abs(int x)
{
   if (x < 0) return -x;
   else       return  x;
}
```

absolute value of a `double` value

```
public static double abs(double x)
{
   if (x < 0.0) return -x;
   else         return  x;
}
```

primality test

```
public static boolean isPrime(int n)
{
   if (n < 2) return false;
   for (int i = 2; i <= n/i; i++)
      if (n % i == 0) return false;
   return true;
}
```

hypotenuse of a right triangle

```
public static double hypotenuse(double a, double b)
{  return Math.sqrt(a*a + b*b);  }
```

harmonic number

```
public static double harmonic(int n)
{
   double sum = 0.0;
   for (int i = 1; i <= n; i++)
      sum += 1.0 / i;
   return sum;
}
```

uniform random integer in [0, n)

```
public static int uniform(int n)
{  return (int) (Math.random() * n);  }
```

draw a triangle

```
public static void drawTriangle(double x0, double y0,
                                double x1, double y1,
                                double x2, double y2 )
{
   StdDraw.line(x0, y0, x1, y1);
   StdDraw.line(x1, y1, x2, y2);
   StdDraw.line(x2, y2, x0, y0);
}
```

Figure 14: func

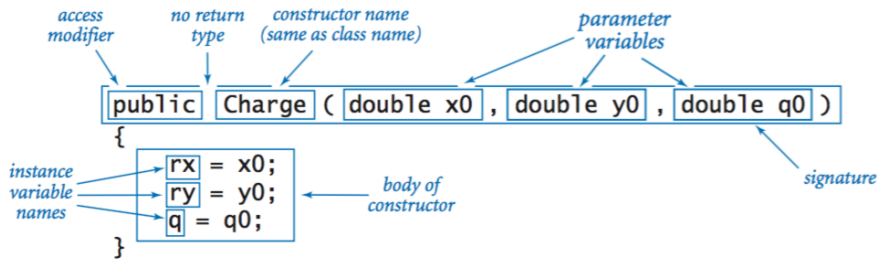


Figure 15: cons

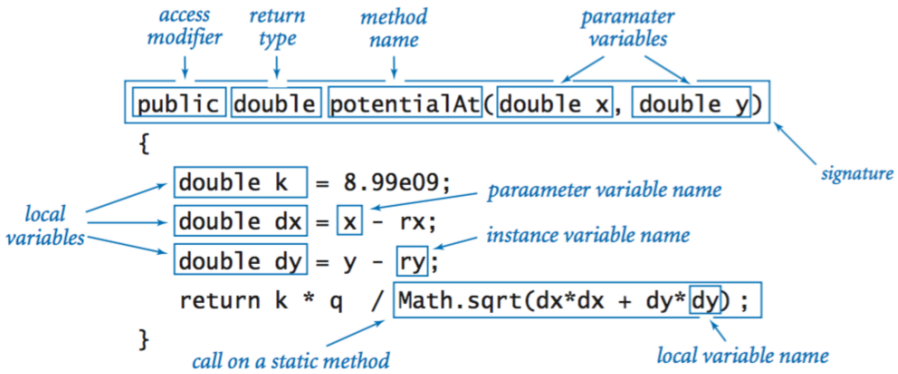


Figure 16: meth