Day 3

COMP 1006/1406A

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
  - 1 was due before class
  - 2 is posted (be sure to read early!)
- a quick look back
  - testing
- test cases for arrays
- object oriented programming (OOP)
  - OOP intro
  - classes and objects
  - constructors
  - encapsulation
  - the Object class
  - inheritance and method overriding
  - inheritance and constructors
  - class attributes

announcements

Office Hours

- Tuesday 8-9pm after the tutorial (2 TAs)
- Wednesday 1-3pm in Jason's office
- Thursday 8-9pm after the tutorial (2 TAs)
- Friday 2-4pm
- Friday 6-8pm

Office Hours

- Tuesday 8-9pm after the tutorial (2 TAs)
  or 5-6pm and 8-9pm with 1 TA each?
- Wednesday 1-3pm in Jason's office
- Thursday 8-9pm after the tutorial (2 TAs)
  or 5-6pm and 8-9pm with 1 TA each?
- Friday 2-4pm
- Friday 6-8pm

last time...

Black box testing

- test the interface (contract)
- use the pre-conditions and post-conditions
- no knowledge of the algorithm/code is known

White box testing

- test the algorithms/code
- you have (some) knowledge of the code

Pay attention to

- border cases
- near border cases
- extreme cases
- typical casses

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\,]}_{1}$$

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\,]}_{1} + \underbrace{[x_1]}_{2^{32}}$$

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\,]}_{1} + \underbrace{[x_1]}_{2^{32}} + \underbrace{[x_1, x_2]}_{2^{64}}$$

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\ ]}_{1} + \underbrace{[x_1]}_{2^{32}} + \underbrace{[x_1, x_2]}_{2^{64}} + \cdots + \underbrace{[x_1, x_2, \ldots, x_{2^{32}}]}_{2^{137,438,953,472}}$$

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\,]}_{1} + \underbrace{[x_1]}_{2^{32}} + \underbrace{[x_1, x_2]}_{2^{64}} + \cdots + \underbrace{[x_1, x_2, \ldots, x_{2^{32}}]}_{2^{137{,}438{,}953{,}472}}$$

- there is about 2^{25} seconds in a year

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\,]}_{1} + \underbrace{[x_1]}_{2^{32}} + \underbrace{[x_1, x_2]}_{2^{64}} + \cdots + \underbrace{[x_1, x_2, \ldots, x_{2^{32}}]}_{2^{137{,}438{,}953{,}472}}$$

- there is about 2^{25} seconds in a year
- it is estimated that there are about 2^{84} atoms in a 70kg person

test cases for arrays

```
public int foo(int[] input);
  // input: an array of integers
  // output: find the biggest number of the input
```

- how many valid inputs are there to this method?

$$\underbrace{[\ ]}_{1} + \underbrace{[x_1]}_{2^{32}} + \underbrace{[x_1, x_2]}_{2^{64}} + \cdots + \underbrace{[x_1, x_2, \ldots, x_{2^{32}}]}_{2^{137,438,953,472}}$$

- there is about 2^{25} seconds in a year
- it is estimated that there are about 2^{84} atoms in a 70kg person
- it is estimated that there are about 2^{272} atoms in the observable universe

test cases for arrays

```
public int foo(String[] input);
  // input: an array of strings
  // output: the total number of characters in all
  //         strings in the array that are not
  //         whitespace
```

what black box test cases do we write for this method?

test cases for arrays

```
public int foo(String[] input);
  // input: an array of strings
  // output: the total number of characters in all
  //         strings in the array that are not
  //         whitespace
```

what black box test cases do we write for this method?

- empty array
  - border case for size of array base case (recursion)

test cases for arrays

```
public int foo(String[] input);
  // input: an array of strings
  // output: the total number of characters in all
  //         strings in the array that are not
  //         whitespace
```

what black box test cases do we write for this method?

- empty array
  - border case for size of array — base case (recursion)
- array with single elements
  - near border case — first recursive case

test cases for arrays

```
public int foo(String[] input);
  // input: an array of strings
  // output: the total number of characters in all
  //         strings in the array that are not
  //         whitespace
```

what black box test cases do we write for this method?

- empty array
  - border case for size of array — base case (recursion)
- array with single elements
  - near border case — first recursive case
- array with several elements
  - typical cases — several recursive calls
  - 5, 10, 50, 100, 5000

test cases for arrays

```
public int foo(String[] input);
  // input: an array of strings
  // output: the total number of characters in all
  //         strings in the array that are not
  //         whitespace
```

what black box test cases do we write for this method?

- empty array
  - border case for size of array — base case (recursion)
- array with single elements
  - near border case — first recursive case
- array with several elements
  - typical cases — several recursive calls
  - 5, 10, 50, 100, 5000
- large arrays
  - extreme cases — many recursive calls
  - 100000, 1000000, 100000000

test cases for arrays

black box test cases for arrays

- two dimensions to test now
  - the size of the array
  - the data inside the array
- test cases for array size
  - smallest valid size (typically empty or singleton array)
  - next smallest valid size
  - typical sizes
  - extreme sizes
- test cases for data in array
  - multiple tests for each array size
  - use black box test cases for the data
  - depends on the given pre/postconditions

test cases for arrays

black box test cases for arrays

- two dimensions to test now
  - the size of the array
  - the data inside the array
- test cases for array size
  - smallest valid size (typically empty or singleton array)
  - next smallest valid size
  - typical sizes
  - extreme sizes
- test cases for data in array
  - multiple tests for each array size
  - use black box test cases for the data
  - depends on the given pre/postconditions
- gets messy really quickly...
  - corner cases!

let's take a break...
for 5 minutes

Object

Oriented

Programming

software engineering

There is no one single best way of writing code

software engineering

There is no one single best way of writing code

- small program (HelloWorld)

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules (classes in Java)
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming
  - a collection of interacting objects

software engineering

There is no one single best way of writing code

- small program (HelloWorld)
  - main method
  - sequence of statements
- bigger programs
  - reduce duplicate code with methods (functions)
  - collect data together (arrays)
- modular programming
  - break up program into modules (classes in Java)
  - each has independent functionality
- procedural programming
  - a sequence of procedure calls that modify the state
  - Basic, Pascal, Fotran, C, Java, C++
- object oriented programming
  - a collection of interacting objects
  - objects have both `state` and `behaviour`

software engineering

- **large** projects are very **complex**

software engineering

- **large** projects are very **complex**
- humans are not perfect...

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?
  - write less code

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?
  - write less code
  - prevent cheating

software engineering

- **large** projects are very **complex**
- humans are not perfect...
  - we make mistakes when solving complex problems
  - we make mistakes because we are lazy
  - we cheat when we can
- how can we reduce the impacts humans have on code?
  - write less code
  - prevent cheating
  - simplify the complexity (**abstraction**)

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
- code is easier to test/debug

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
- code is easier to test/debug
- code is easier for others to read

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
  - saves us time developing/maintaining/upgrading code
- code is easier to test/debug
  - saves us time developing/maintaining/upgrading code
- code is easier for others to read
  - saves us time developing/maintaining/upgrading code

procedural programming

- data structures hold data (state of the program)
- a `main` method provides coarse-grain control flow for program
- program is a sequence of procedure calls that modify the state
  - each procedure should do one thing (modularization)

why might we want to program like this?

- code is easier for us to read
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity
- code is easier to test/debug
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity
- code is easier for others to read
  - saves us time developing/maintaining/upgrading code
  - saves us money, time and our sanity

object oriented programming (OOP)

a different approach to programming that focuses on objects interacting with each other (passing messages to each other)

Four main principles of object oriented programming:

- **abstraction**
  - allows us use manage complexity
  - allows us to use objects without knowing exactly how they work
- **encapsulation**
  - allows us to model very complex real-world problems nicely
  - is a mechanism to allow for abstraction
- **inheritance**
  - promotes code sharing and re-usability (write less code!)
  - allows us to exploit natural hierarchical structure
- **polymorphism**
  - simplifies code understanding
  - allows us to standardize method names

object oriented programming (OOP)

- **abstraction**
  - classes implement abstract data types (**ADT**s)
  - classes have an interface and an implementation

- **encapsulation**
  - encapsulation has two meanings

  1) objects have both `state` and `behaviour`
  2) objects hide their internal structure (**information hiding**)

- **inheritance**
  - classes are able to inherit state and behaviour from other classes

- **polymorphism**
  - objects can act like other objects. `dynamic binding` allows objects to determine which methods to use at runtime.
  - methods can have the same name. `early binding` determines which method should be executed (`overloading`)

Note: OOP shines in BIG projects
(don't be discouraged if it seems like a lot of work at first)

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

int ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
`+, -, *, /, %, <, >, <=, ...`

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

int ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
`+, -, *, /, %, <, >, <=, ...`

String ▸ sequence of zero or more characters
+ (concatenation), `toUpper()`, etc

classes and objects

- **data type** ▸ is a set of values and a set of operations defined on those values
- **int** ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
  `+, -, *, /, %, <, >, <=, ...`
- **String** ▸ sequence of zero or more characters
  + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types

classes and objects

- **data type** ▸ is a set of values and a set of operations defined on those values
- **int** ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
  `+, -, *, /, %, <, >, <=,...`
- **String** ▸ sequence of zero or more characters
  + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types
- Many non-primitive data types are available (String, Date, etc)

classes and objects

- **data type** ▸ is a set of values and a set of operations defined on those values
- **int** ▸ the integers $-2,147,483,648 \rightarrow 2,147,483,647$
  `+, -, *, /, %, <, >, <=, ...`
- **String** ▸ sequence of zero or more characters
  + (concatenation), `toUpper()`, etc

- Java has 8 primitive data types
- Many non-primitive data types are available (String, Date, etc)
- Java allows us to make our own data types
  - each `class` is a new data type
  - specifies data and operations on the data

classes and objects

data type
- is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (how many cookies made?)
    - class attributes / static attributes / class fields

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (how many cookies made?)
    - class attributes / static attributes / class fields
  - can have its own functionality (bottle opener?)
    - class methods / static methods (functions)

classes and objects

data type ▸ is a set of values and a set of operations defined on those values

- a **class** is a data type (the cookie cutter)
  - specifies what data can be stored
    - instance attributes
  - defines operations on that data
    - instance methods
  - can have its own data (how many cookies made?)
    - class attributes / static attributes / class fields
  - can have its own functionality (bottle opener?)
    - class methods / static methods (functions)

- an **object** is an instantiation of a class (the cookie)
  - holds data (the **state** of the object)
    - instance attributes
  - has operations built-in to it (the **bahaviour** of the object)
    - instance methods

anatomy of a class

```
public class MyClass{
    /* class attributes */
    public static int count;
    public static final double PI = 3.145;

    /* instance attributes */
    public int a;
    private String s;

    /* constructors */
    public MyClass(){ ... }
    public MyClass(int x){ ... }

    /* instance methods */
    public int addOne(){...}

    /* class methods */
    public static void main(String[] args){...}
}
```

anatomy of a class

```
public class MyClass{
    /* class attributes */
    public static int count;
    public static final double PI = 3.145;

    /* instance attributes */
    public int a;
    private String s;

    /* constructors */
    public MyClass(){ ... }
    public MyClass(int x){ ... }

    /* instance methods */
    public int addOne(){...}

    /* class methods */
    public static void main(String[] args){...}
}
```

⟵ defines the STATE of the class

⟵ defines the STATE of each object

⟵ initialization code when creating an object

⟵ defines the BEHAVIOUR of objects

⟵ defines the BEHAVIOUR of the class

class as a container

```
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

class as a container

```
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

- simple aggregation of data (a container)
  you should have seen this in 1005/1405
- data stored can be different types
- array was good for collecting together data of the same type
  (doesn't work for different data though)
- this is essentially a `record` in Pascal or a `struct` in C

class as a container

```
public class Student{
    /* instance attributes */
    public String name;
    public int id;
}
```

- declare variable of type `Student` like any other variable
  - `Student s;`
- instantiate the actual object with `new` operator and constructor
  - `s = new Student();`
- access attributes with `dot` operator and name of attribute
  - `s.name = "fig";`
- in an array you access the data by its position (index) in the array
  - `names[2] = "date";`

constructors

a **constructor** contains code that is executed when an object of the class is instantiated (created)

for example, int the declaration

```
Student s = new Student();
```

`Student()` is a constructor for the Student class

we can think of constructors as creation or initialization methods, but

- constructors must have the same name as the class
- constructors have no return value (not even `void`)
- constructors can only be called with the `new` operator (and one other time that we will see soon)
- constructors are NOT methods (although they are similar)

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructor */
    public Student(){
        name = "none";
        id = -1;
    }
}
```

an example of a simple constructor

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructor */
    public Student(String name, int id){
            name = name;
            id = id;
    }
}
```

constructors can have input parameters

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructor */
    public Student(String name, int id){
        this.name = name;
        this.id = id;
    }
}
```

constructors can have input parameters

- Java keyword `this`
  - a reference to the current object
  - used in constructors and instance methods
  - has other uses we'll discuss soon
- `this` is needed here because attributes `name` and `id` are not in scope (the input parameters `name` and `id` are in scope)

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructor */
    public Student(String nameInit, int idInit){
            name = nameInit;
            id = idInit;
    }
}
```

- `this` is not needed here

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructor */
    public Student(String nameInit, int idInit){
        this.name = nameInit;
        this.id = idInit;
    }
}
```

- `this` is not needed here
- you can still use it though!
- I will often include it in constructors for this course

constructors

```
public class Student{
    public String name;
    public int id;

    /* constructors */
    public Student(String nameInit, int idInit){
        this.name = nameInit;
        this.id = idInit;
    }

    public Student(String nameInit){
        this.name = nameInit;
        this.id = -1;
    }
}
```

Java allows method and constructor **overloading**

- can have as many constructors as is useful

signatures and overloading

A method or constructor **signature** consists of

- the **name** of the method or constructor
- the **input parameters** of the method or constructor
  (number, type and order)

Java identifies methods and constructors by their signatures.
This allows for method and constructor **overloading**.

- overloading allows multiple methods/constructors to have the same name as long as their signatures are different
  (return types do not matter!)
- this is very useful! Consider the `println` method
  https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html

constructors

```
public class Student{
    public String name;
    public int id;




    public static void main(String[] args){
        Student s = new Student();
    }
}
```

A class does not need to have a specified constructor to work.

constructors

```
public class Student{
    public String name;
    public int id;

    public Student(){
    }

    public static void main(String[] args){
        Student s = new Student();
    }
}
```

A class does not need to have a specified constructor to work.

- Java automatically provides a zero argument default constructor if none are specified

constructors

```
public class Student{
    public String name;
    public int id;

    public Student(){
    }

    public static void main(String[] args){
        Student s = new Student();
    }
}
```

A class does not need to have a specified constructor to work.

- Java automatically provides a zero argument default constructor if none are specified
- Java only does this if NO constructors are specified

  we need to be careful! dangers to come...

constructor chaining

a constructor can only be called in two situations in Java

- when instantiating an object with `new`
  - `House h = new House("123 Sesame Street");`
- from within a constructor of the same class

calling a constructor from within another constructor of the same class is called **constructor chaining**

- the keyword `this` can also be used to call constructors

constructor chaining

calling a constructor from within another constructor of the same class is called **constructor chaining**

consider three constructors

```
public Ball()
  // initialize ball at x=y=0 with speed dx=dy=0

public Ball(int x, int y)
  // initialize ball with given x,y and speed dx=dy=0

public Ball(int x, int y, int dx, int dy)
  // initialize ball with given coordinates and speed
```

constructor chaining

```
public Ball(){
  this.x = 0;
  this.y = 0;
  this.dx = 0;
  this.dy = 0;
}

public Ball(int x, int y){
  this.x = x;
  this.y = y;
  this.dx = 0;
  this.dy = 0;
}

public Ball(int x, int y, int dx, int dy){
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
}
```

constructor chaining

```
public Ball(){
  this.x = 0;
  this.y = 0;
  this.dx = 0;
  this.dy = 0;
}

public Ball(int x, int y){
  this(x,y,0,0);




}

public Ball(int x, int y, int dx, int dy){
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
}
```

constructor chaining

```
public Ball(){
  this(0,0,0,0);   // or this(0,0);




}

public Ball(int x, int y){
  this(x,y,0,0);




}

public Ball(int x, int y, int dx, int dy){
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
}
```

constructor chaining

when using this to call another constructor from within a constructor, it **must** be the very first line of the constructor

- you can have more code after the call if needed

```
public Ball(){
  this(0,0,0,0);   // or this(0,0);
  System.out.println("zero argument constructor");
}

public Ball(int x, int y){
  this(x,y,0,0);
  System.out.println("two argument constructor");
}

public Ball(int x, int y, int dx, int dy){
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
  System.out.println("four argument constructor");
}
```

let's take a break...
for 5 minutes

encapsulation

encapsulation refers to two ideas

- classes and objects have both state and behaviour
- the internal details of the data is hidden

We'll look at the second idea more now

- often called **information hiding**
- related to idea of **separation of concerns**
  - actual code and how you use the code are independent
- access to data is restricted
  - **getter** or **accessor** allows us to see the data
  - **setter** or **mutator** allows us to change the data
  - not all data will be visible and not all data will be allowed to be modified
- why would we want to do this?

encapsulation

encapsulation refers to two ideas

- classes and objects have both state and behaviour
- the internal details of the data is hidden

We'll look at the second idea more now

- often called **information hiding**
- related to idea of **separation of concerns**
  - actual code and how you use the code are independent
- access to data is restricted
  - **getter** or **accessor** allows us to see the data
  - **setter** or **mutator** allows us to change the data
  - not all data will be visible and not all data will be allowed to be modified
- why would we want to do this?
  - what happens if you change how you store your data?
  - if someone has access to a variable, will they modify it?

encapsulation

```
public class Student{
    public String name;
    public int id;

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

⟵ anyone can access/modify these attributes

encapsulation

```
public class Student{
    private String name;
    private int id;
```

⟵ attributes can only be accessed from within the class

```
    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

encapsulation

```
public class Student{
    private String name;         ⟵ attributes can only be accessed
    private int id;                             from within the class

    /* getter - accessor */
    public String getName()
        {return this.name;}

    /* setter - mutator */
    public void setName(String newName)         ⟵ return void
        {this.name = newName;}

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

Would we also have a `getID()` and `setID(int id)`?

encapsulation

```
public class Student{
    private String name;
    private int id;

    /* getter - accessor */
    public String getName()
        {return this.name;}

    /* setter - mutator */
    public String setName(String newName)        ⟵ return String
        { String old = this.name;
          this.name = newName;
          return old; }

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

Might return the old value

encapsulation

```
public class Student{
    private String name;
    private int id;

    /* getter - accessor */
    public String getName()
        {return this.name;}

    /* setter - mutator */
    public Student setName(String newName)          ⟵ return self
        { this.name = newName;
          return this;}

    public Student(String name, int id)
        {this.name = name; this.id = id; }
}
```

Why would we return this?

method chaining

calling a method on the returned object of another method call is called **method chaining**

for example, consider the getter/setter from the last slide

```
public String getName()
    { return this.name; }

public Student setName(String newName)
    { this.name = newName;
      return this; }
```

The following would be valid

```
Student s = new Student("Cat", 2);
char c = s.setName("GalADriEl").getName().toLowerCase().charAt(4);
```

Java's Object class

```
public class Object{
    /* no attributes */

    /* single constructor */
    public Object(){}

    /* 11 methods */
    public String toString(){...}
    public int hashCode(){...}
    public boolean equals(Object obj){...}
    ...
}
```

- `java.lang.Object`
- this is Java's root class
  (its basic non-primitive type)

inheritance

Let's run the following simple class;

```
public class Student {
    /* attributes */
    public String name;
    public int id;

    public static void main(String[] args){
        Student s = new Student();
        System.out.println(s);
        System.out.println(s.toString());
        System.out.println(s.hashCode);
        System.out.println(s.equals(s));
    }
}
```

inheritance

Let's run the following simple class;

```
public class Student {
    /* attributes */
    public String name;
    public int id;

    public static void main(String[] args){
        Student s = new Student();
        System.out.println(s);
        System.out.println(s.toString());
        System.out.println(s.hashCode);
        System.out.println(s.equals(s));
    }
}
```

Why does this work?

inheritance

```
public class Student{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

when you compile this class it is automatically modified to inherit from the `Object` class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public attributes from the parent class
  - we get all public methods from the parent class
  - we get none of the constructors

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public attributes from the parent class
  - we get all public methods from the parent class
  - we get none of the constructors
- we say that **Student is an Object**
  - this is the "is-a" relationship
- we say that **Student has a String**
  - this is the "has-a" relationship
  - this is class composition (not inheritance)

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- a class can only have one parent class
- every class, except `Object`, has exactly one parent class
- we get a hierarchy of classes
    - a family tree of classes

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

we say that

- Student is a child class of Object
- Student is a subclass of Object
- Student is a derived class of Object
- Student is a descendent of Object

we say that

- Object is a parent class of Student
- Object is a super class of Student
- Object is a ancestor of Student

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

what do we get from `Object`?

- `toString()`
- `hashCode()`
- `equals(Object o)`
- eight other methods that we will most likely not use

Are these useful?

`Object`'s `equals` method

`public boolean equals(Object obj)`

- checks if both `this` and `obj` are the same
- returns `this == obj`
- this is almost certainly not what we want! (why?)

inheritance - method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;

    @Override                                   ⟵ @Override is an annotation
    public String toString(){
        return this.name + ", " + this.id;
    }
}
```

method `overriding` allows us to redefine a parent's (or grandparent's) method definition. which method is executed?

- Java first looks in current class
- if method is not defined, look at parent class
- if method is not defined, look at parent class
- ...
- get method from `Object`

inheritance - method overriding

rules for **method overriding**

- signature must be identical
- return type must be the same or more restrictive
  - same if primitive
  - more restrictive related type of object (inheritance)
- modifiers must be the same
  - or less restrictive

inheritance - method overriding

let's see some examples...

inheritance and constructors

by default, the first thing that any constructor we write does is call the zero argument constructor of its parent class

if want another constructor called from the parent we need to explicitly call it using the `super` keyword

we cannot call both `super` and `this` in a constructor (as each of them must be on the very first line of they are explicitly used)

inheritance and constructors

let's see some examples...

let's take a break...
for 5 minutes

class attributes and methods

things that are static belong to the class and not objects

```
public class Box{
    private static int secret;
    public static int xStatic;
}
```

what can we say about static attributes?

- they exist even if an object of the class does not (think of Math)
- public attributes can be accessed/modified by any object or class in the program
  - usually not a good idea unless they are constants (final)
  - final is tricky though... must be careful with it
- private attributes are are only accessible within the class (information hiding!)

the `final` modifier

things that are `final` cannot be changed once they are defined

```
public final int x = 3;
public final String str = "cat";
public final Student s = new Student("dog", 4);
public final int[] numbers = {1,3,5,7,9};
```

Which are valid/invalid?

- `System.out.println(x);`
- `x = 4;`
- `str = "dog";`
- `s.setName("eel");`
- `numbers[2] = 100;`

the final modifier

things that are `final` cannot be changed once they are defined

```
public final int x = 3;
public final String str = "cat";
public final Student s = new Student("dog", 4);
public final int[] numbers = {1,3,5,7,9};
```

Which are valid/invalid?

- `System.out.println(x);` ✓
- `x = 4;` ✗
- `str = "dog";` ✗
- `s.setName("eel");` ✓
- `numbers[2] = 100;` ✓

Remember that ONLY the value in the variable is held constant.
Final primitive data types and strings are constants. For reference data types, the data may change.

class attributes and methods

things that are `static` belong to the class and not objects

```
public class Box{
    public static int xStatic;
    public int x;
}
```

what can we say about `static` methods?

class attributes and methods

things that are `static` belong to the class and not objects

```
public class Box{
    public static int xStatic;
    public int x;
}
```

what can we say about `static` methods?

- what if they are `public`?
- what if they are `private`?
- what if they are `final`?
    - to come...