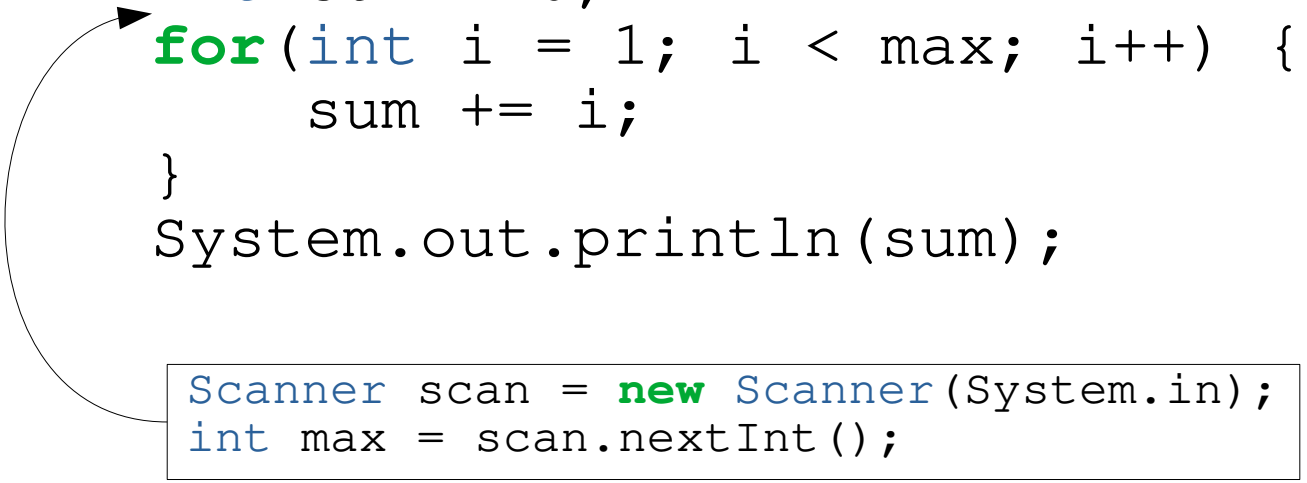
Lecture 8
Exam preparation

Exam tips & tricks

- Q: How are the exams graded?
- A: $\sum_{i=1}^{qs} max(0, maxpoints_i - deductions_i)$
  - Deductions are made for (including but not limited to):
    - Wrong answers
    - Type errors
    - Significantly overcomplicated solutions
  - Deductions are **NOT** made for (unless explicitly specified):
    - Minor syntax errors
    - Lack of comments
    - Lack of error handling
    - Slow solutions

Exam tips & tricks

- Q: What do I do if I realize that I need to insert some more code in the middle of my solution?
- A: Draw a box with your code, with an arrow pointing to where you want to insert it.



```
int sum = 0;
for(int i = 1; i < max; i++) {
    sum += i;
}
System.out.println(sum);
```

```
Scanner scan = new Scanner(System.in);
int max = scan.nextInt();
```

Exam tips & tricks

- Q: Do I need to include `X`?
- `X` = `import` statements: no
- `X` = comments: no
- `X` = error checking: only if explicitly asked
- `X` = `{` braces `}`: yes, but we'll be lenient about it
- `X` = indentation: **YES!**
- `X` = assumptions (if question is unclear): **YES!**

Exam tips & tricks

- Read through the entire exam before you start working!
  - This will help you plan your time
- Write down any questions about the exam on a scrap paper!
  - This will help you remember to ask them when I drop by at 9 AM
  - My next visit isn’t until 11 AM!

Exceptions

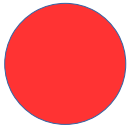
An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
(Definition from https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html)

We can **catch** exceptions in order to handle them gracefully.

If an exception happens and is not caught, then the program crashes.

Catching Exceptions

```
try {
  block 0
} catch (ExceptionType1 varName1) {
  block 1
} catch (ExceptionType2 varName2) {
  block 2
} … catch (ExceptionTypen varNamen) {
  block n
} finally {
  block n+1
}
```

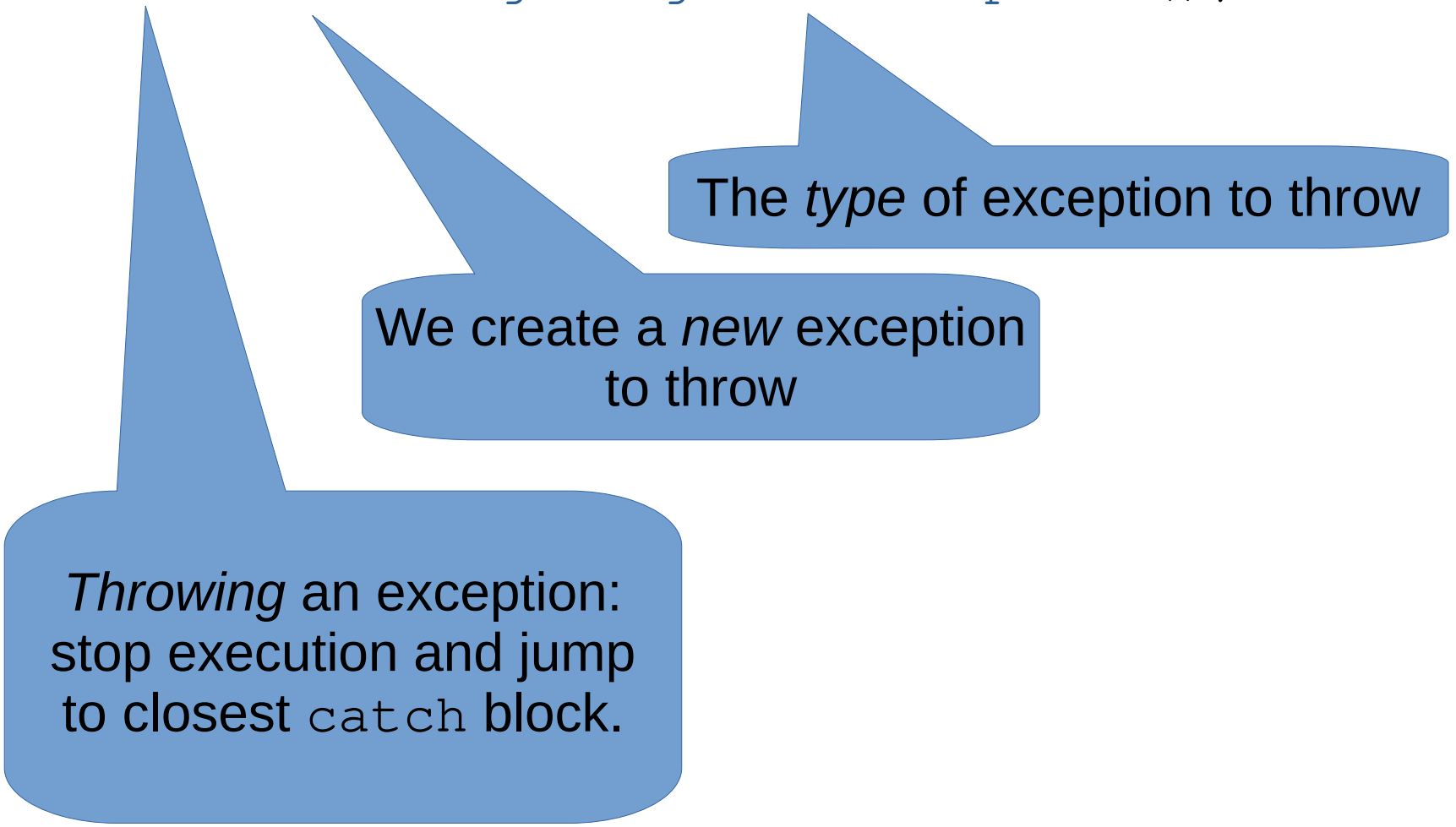


1) Block 0 is executed.
2) If no exception occurs: block n+1 is executed.
3) If an exception of type `ExceptionType` occurs while executing block 1, then the exception is stored in the variable (“caught”), and block 2 is executed, then block n+1 is executed.
4) If an exception of another type occurs, block n+1 is executed, then the exception is thrown to the calling method. (If this is in `main`: the program crashes.)

The part **`finally`** `{` block n+1 `}` is optional

Throwing Exceptions

```
throw new IllegalArgumentException();
```



Exception control flow

- Prerequisite knowledge: Ariadne and Theseus

Exception control flow

- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```

Exception control flow

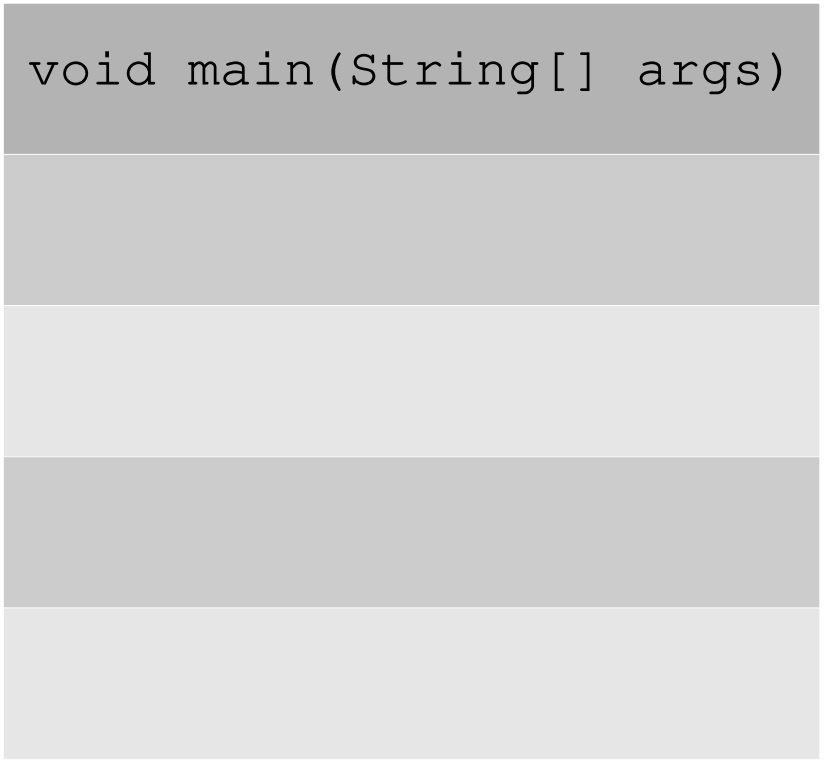
- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```



* = contains a `catch` block

Exception control flow

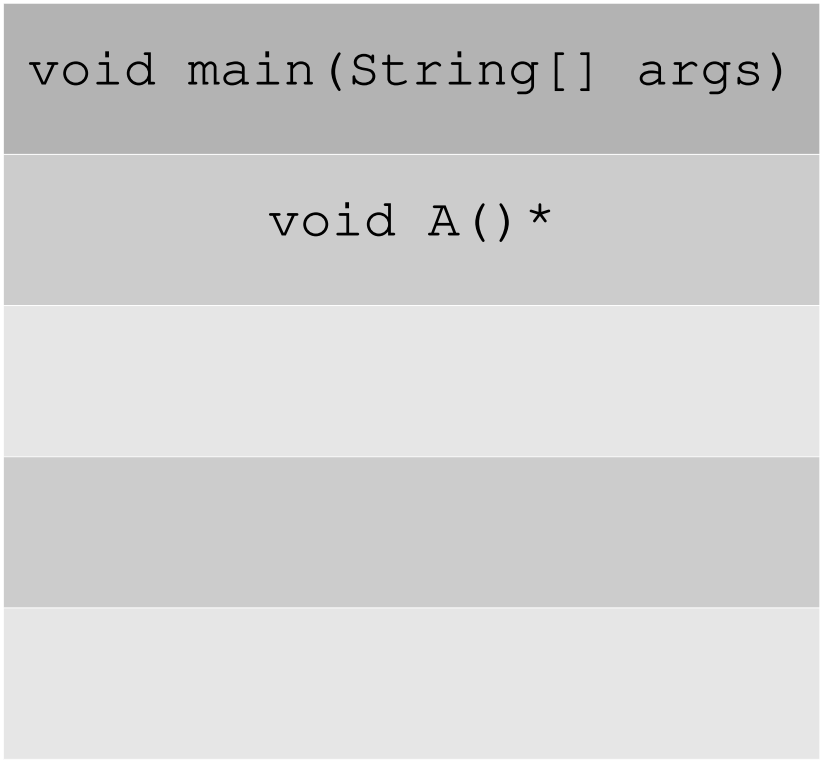
- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```



* = contains a `catch` block

Exception control flow

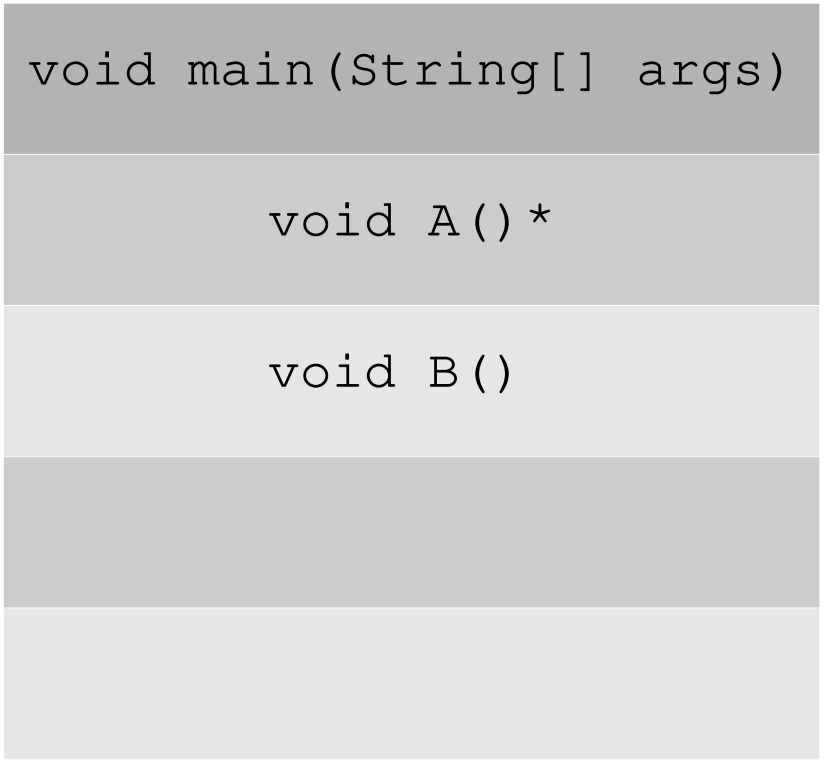
- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```



* = contains a `catch` block

Exception control flow

- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```

void main(String[] args)
void A()*
void B()
void C()*

* = contains a `catch` block

Exception control flow

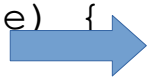
- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```



void main(String[] args)
void A()*
void B()
void C()*
void D()

* = contains a `catch` block

Exception control flow

- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```

void main(String[] args)
void A()*
void B()
void C()*

* = contains a `catch` block

Exception control flow

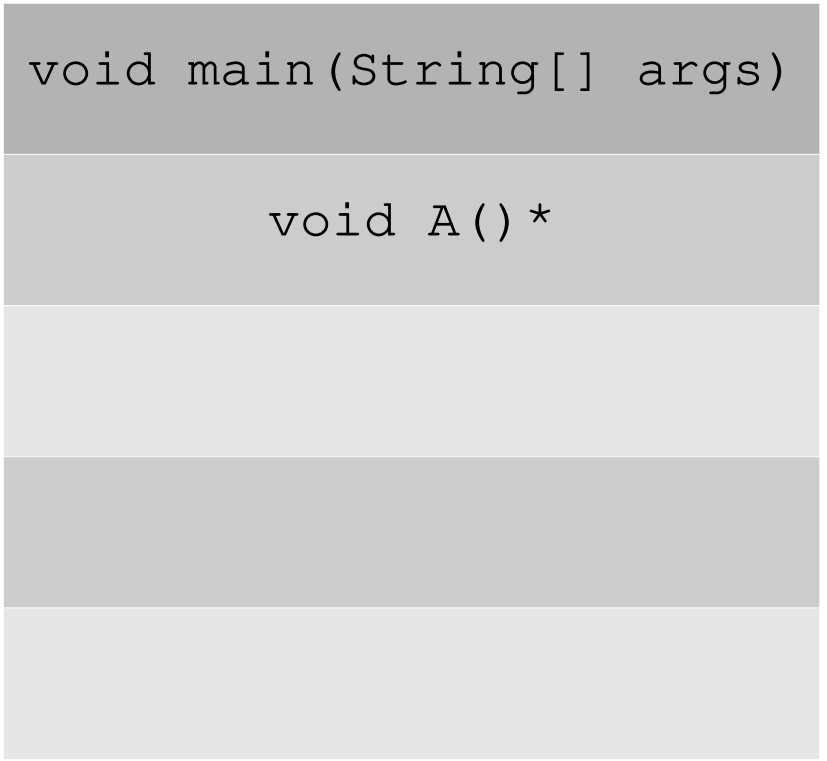
- "Closest" `catch` block?

```
public static void A() {
  try {
    B();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```



* = contains a `catch` block

Exception control flow

- "Closest" `catch` block?

```
public static void A() {
```

Closest `catch` block = the `catch` block we arrive at first when "following the thread out of the labyrinth".

```
{

public static void B() {
  C();
}

public static void C() {
  try {
    D();
  } catch (IllegalStateException e) {
    // do something useful here
  }
}

public static void D() {
  throw new IllegalStateException();
}
```

void main(String[] args)
void A()*
void B()
void C()*
void D()

* = contains a `catch` block

Throwing Exceptions

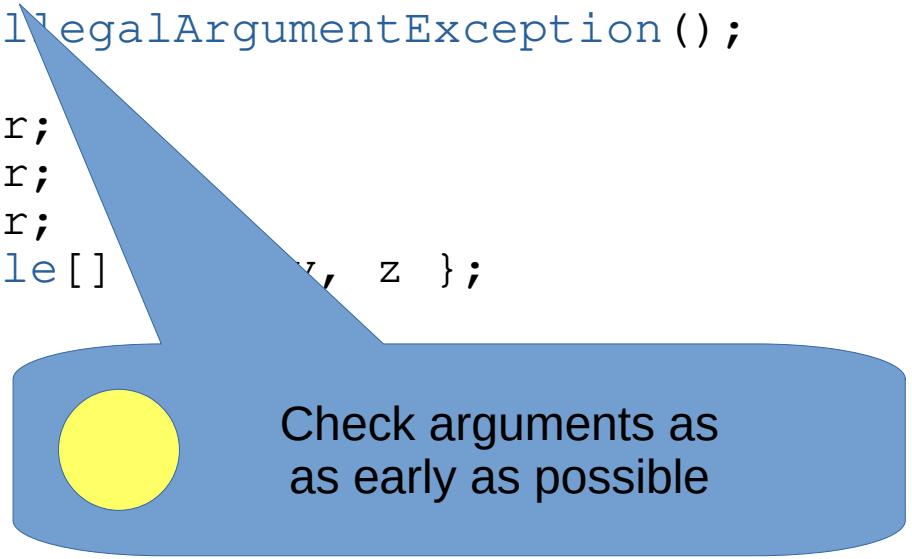
- Common use case: checking method arguments

```
public static double[] divide(double[] a, double divisor) {
    double x, y, z;
    if(Math.abs(divisor) < 0.0001) {
        throw new IllegalArgumentException();
    }
    x = a[0]/divisor;
    y = a[1]/divisor;
    z = a[2]/divisor;
    return new double[] { x, y, z };
}
```

Throwing Exceptions

- Common use case: checking method arguments

```
public static double[] divide(double[] a, double divisor) {
    double x, y, z;
    if(Math.abs(divisor) < 0.0001) {
        throw new IllegalArgumentException();
    }
    x = a[0]/divisor;
    y = a[1]/divisor;
    z = a[2]/divisor;
    return new double[]       y, z };
}
```



References

- Java has both *primitive types* and *reference types*.
- *Primitive types* represent *single values*:
  - `int`, `char`, `double`, etc.
- *Reference types* represent *collections of values*:
  - `String` (a collection of letters)
  - `int[]` (a collection of ints)
  - `Pony` (a collection of two strings)
  - `HangMan` (a collection of a HangWord and an `int`)

References

- Values of primitive types are stored directly in their variables.
  - `int x = 42;` ← Here `x` literally contains the value 42
- Values of reference types are called *objects*, and are stored in *another part of memory*.
- Variables of reference types only contain a *pointer* to the memory where the actual values are stored.
  - `Pony p = new Pony("Aristotle");`
  - Here `p` only contains a pointer – or *reference* – to the memory address where the actual Aristotle pony is stored!
- Why? Because objects can be *very large*, which makes copying them around in memory *very inefficient*.
- A *reference* to an object only takes 8 bytes of memory, so it is very efficient to pass around.

References

- Multiple reference variables can refer to the same object.

```
Pony ponyA = new Pony("Aristotle");
Pony ponyB = ponyA;
```

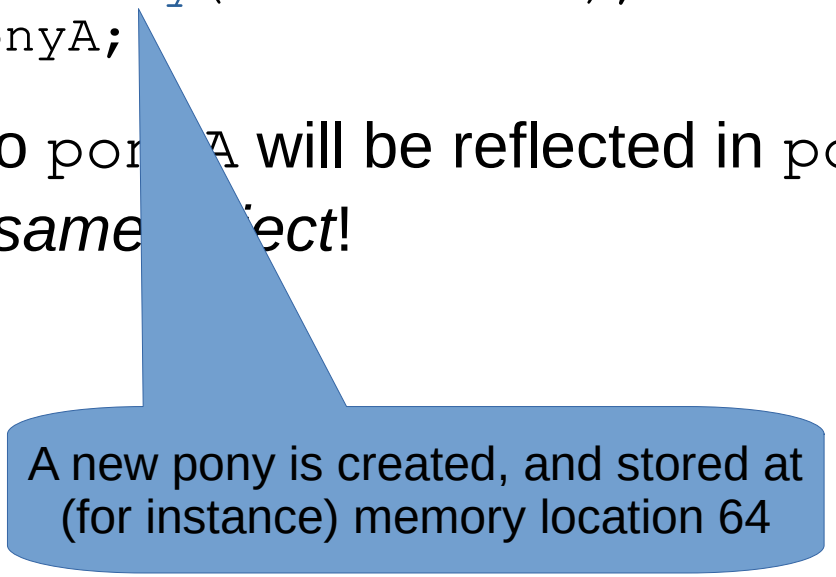
- Changes made to `ponyA` will be reflected in `ponyB` – because they refer to *the same object*!

References

- Multiple reference variables can refer to the same object.

```
Pony ponyA = new Pony("Aristotle");
Pony ponyB = ponyA;
```

- Changes made to `ponyA` will be reflected in `ponyB` – because they refer to *the same object*!

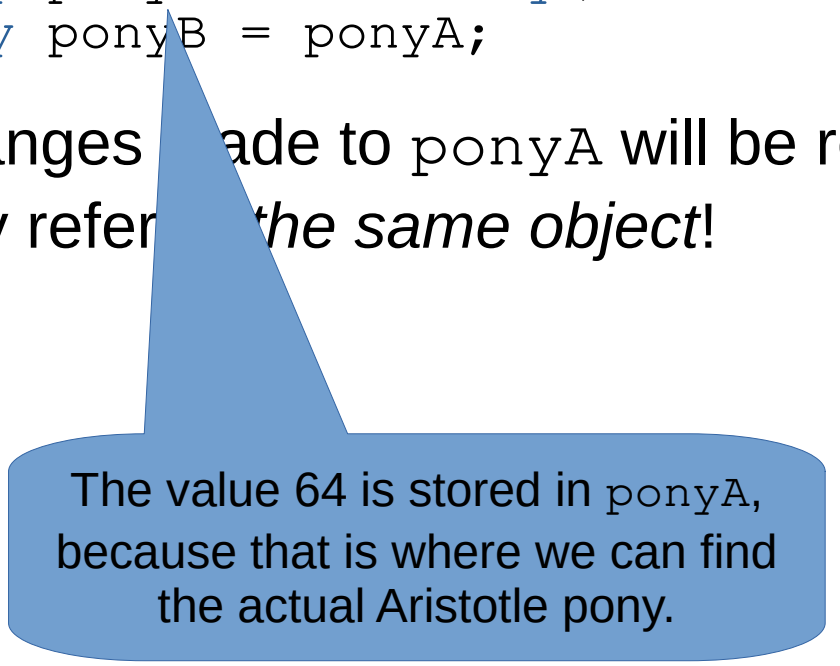


References

- Multiple reference variables can refer to the same object.

```
Pony ponyA = new Pony("Aristotle");
Pony ponyB = ponyA;
```

- Changes made to `ponyA` will be reflected in `ponyB` – because they refer to *the same object*!



References

- Multiple reference variables can refer to the same object.

```
Pony ponyA = new Pony("Aristotle");
Pony ponyB = ponyA;
```

- Changes made to `ponyA` will be reflected in `ponyB` – because they refer to *the same object*!

The value 64 is copied from `ponyA` to `ponyB`.
Now both refer to the same pony!

References

- Equality operator only compares *the references*!
- Two identical objects are **not** equal if they are stored at different memory locations, according to the == operator.

```
Pony ponyA = new Pony("Aristotle");
Pony ponyB = new Pony("Aristotle");
System.out.println(ponyA == ponyB); // prints “false”
```

- To compare objects, we must use the `equals` method!

```
String a = "hello";
String b = String.format("hello");
System.out.println(a == b); // prints “false"
System.out.println(a.equals(b)); // prints “true”
```

- General idea: compare objects element for element.
- See lecture 6 for details.

References

- Sometimes we want to make an identical copy of an object.
  - Usually because we want to be sure other parts of the program can't make unexpected changes to it.
  - But also because we want to make changes in our copy without affecting other parts of the program.
- For our own classes: create a *copy constructor*!

```
public Pony(Pony original) {
    this.name = original.name;
    this.skill = original.skill;
}
```

- For arrays: copy element by element, or use `clone` method!

```
int[] arr1 = {1, 2, 3};
int[] arr2 = arr1.clone();
arr1[0] = 100; // does not affect arr2
```

References

- If your object or array contains other objects, you need to make a *deep copy*!

```
public static Pony[] copyPonies(Pony[] ponies) {
    Pony[] clones = new Pony[ponies.length];
    for(int i = 0; i < ponies.length; i++) {
        clones[i] = new Pony(ponies[i]);
    }
    return clones;
}
```

- You need to make a deep copy of everything that's not:
  - a primitive type; or
  - an *immutable* class.

References

- An *immutable class*:
  - has only `final` fields; and
  - has only fields of primitive types OR other immutable classes.
- Intuition: an object of an immutable class can’t be changed after it’s created.
- `String` is immutable: all its operations return a new string, none modify the existing object.
- Writing immutable (or partially immutable) classes when possible is good coding style!

Constructors

- A constructor is a special method which initializes an object.
- Returns a reference to the newly created object.
- To call the two argument constructor of the `Pony` class and store the resulting pony in a variable:

```
Pony myPony = new Pony("Socrates", "philosophy");
```

- The purpose of the constructor is to ensure that the newly created object is ready to use.

Constructors

- The creation of an object:
  - The programmer calls `new SomeClass(...)`
  - Java allocates memory for the object's fields
  - Java calls the specified constructor for `SomeClass`
  - The code in the constructor is executed
  - A reference to the newly created object (`this`) is returned

Constructors

- The creation of an object:
  - The programmer calls `new SomeClass(...)`
  - Java allocates memory for the object's fields
  - Java calls the specified constructor for `SomeClass`
  - The code in the constructor is executed
  - A reference to the newly created object (`this`) is returned

This is the only part you (i.e. the person who implements the class) have control over!

Constructors

- Constructors are not magic!
- The following will have **no effect**:

```
public Pony(String name) {
    new Pony(name, "no particular skill");
}
```

- You are creating a *new* pony and discarding it right away!
- Just as if you did the same in any other method.
- Instead, you need to either *chain* to another constructor, fill in the object's fields yourself, or both.

What's `this`?

- `this` is a reference to the object we're currently executing a method or constructor on.
- We can access fields and other methods on *the same object* using `this.someMethod(arg1, arg2, …)` and `this.someField` respectively.
- However, `this` can be omitted!
  - `this.someMethod(arg1, arg2, …)` and `someMethod(arg1, arg2)` are equivalent!
  - So are `this.someField = 0` and `someField = 0`

What's `this`?

- There are two cases where `this` is mandatory:
  - Disambiguating between local variables and fields:

```
public class Pony {
    private String name;
    …
    public Pony(String name) {
        this.name = name;
    }
}
```

  - When we want to pass a reference to our object

```
public class Pony {
    private String name;
    …
    public void addToArray(Pony[] ponies, int i) {
        ponies[i] = this;
    }
}
```

What's `this`?

- `this` can also refer to *another constructor of the same class*, when used in a constructor.

```
public class Pony {
    private String name;
    private String skill;

    public Pony(String name, String skill) {
        this.name = name;
        this.skill = skill;
    }

    public Pony(String name) {
        this(name, "eating");
    }
}
```

- This is called "constructor chaining" - very handy to keep DRY!

To `static` or not to `static`

- `static` methods and variables belong to *the class itself*, not objects of the class.
  - There is only a single copy of each `static` method or variable in your whole application.
  - They are accessed using *the class itself*: `ClassName.method()`
- Non-`static` methods and variables (fields) belong to *objects of the class.*
  - They can only be accessed using an object of the class: `object.method()`
  - There is one copy of each non-`static` method and field of a class for each object of that class.
- Non-static methods have access to:
  - `this`
  - Other non-`static` methods and fields
- `static` methods do not.

`public` vs `private`

- When a method or field is `public`, it can be accessed from other classes.
  - The methods we make `public` define how we expect the user of our class to interact with it.
- When `private`, it can only be accessed from the class in which it is defined.
  - Always make fields `private`
  - Make helper methods `private`

Boolean expressions

- Logical expressions consist of one or more *truth values* connected by conjunction (`&&`), disjunction (`||`) or negation (`!`).
- Logical expressions are themselves truth values.
- Truth values have the type `boolean`.
- Examples of truth values:
  - Boolean constants: `true`, `false`
  - Comparisons: `x == y`, `z != null`, `a > b`
  - Logical expressions:
    - `(c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')`
    - `!x`

Boolean expressions

- `if`, `while`, etc. accept **ANY** boolean expression
  - `if(x == 5)`
  - `while(y)`
    (where `y` has type boolean)
  - `if(someMethod(x, y, z))`
    (where `someMethod` has return type `boolean`)
  - `if(x == 5 && !y)`
- It does **NOT** have to be a comparison!
- Don't do `if(x == true)`, do `if(x)` instead.
- Don't do `if(x == false)`, do `if(!x)` instead.

Boolean expressions

- There is nothing “magical” about the condition used for `if`, `while`, etc.
- Don’t:

```
if(some boolean expression) {
    return true;
} else {
    return false;
}
```

- Do:

```
return some boolean expression;
```

String.format

- `String String.format(String fmt, type1 arg1, type2 arg2, …)`
- `fmt` is a plain string which may or may not contain *format specifiers*.
- The most basic format specifiers:
  - `%s` – a string
  - `%d` – an integer
  - `%f` – a decimal number
- Format specifiers can be prefixed with a number
  - `%10s` – a string, left-padded with spaces to at least 10 chars
  - `%5d` – an integer, left-padded with spaces to at least 5 chars
- %f can also be prefixed with a dot and the number of digits of precision to use
  - `%.0f` – a decimal number rounded to the closest whole number
  - `%.2f` – a decimal number rounded to two digits of precision
  - `%10.2f` – a decimal number rounded to two digits of precision, *and* left-padded with spaces to at least 10 chars

`import` vs. `import static`

- `import` brings a class into scope
  - We still need to explicitly name the class to use it
  - **`import`** `java.util.Scanner;` lets us create and use `Scanner` objects:

```
Scanner scan = new Scanner(System.in);
System.out.println(scan.nextInt());
```

  - **`import`** `javax.swing.JOptionPane;` lets us call static methods on `JOptionPane` by *explicitly* referencing the class:

```
JOptionPane.showMessageDialog(null, "Hej!");
```

- `import static` brings *all static members of a class* into scope
  - We *don't* need to name the class to use them:

```
import static javax.swing.JoptionPane.*;
...
showMessageDialog(null, "Hej!");
```

Single quotes vs. double quotes

- *Exactly one* letter within *single quotes* denotes a single character, and has type `char`
  - `char c = 'a';`
- Zero or more letters within *double quotes* denotes a string, and has type `String`
  - `String s = "Hello!";`
- Both may contain escaped characters
  - String s = "Hello\nWorld";
  - char c = '\n';

Scanner

- Lets us read structured data from a text source.
- So far, we've used *standard input*:

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- But we can also use it to read from a string:

```
String myString = "10 15 hello";
Scanner s = new Scanner(myString);
System.out.println(s.nextInt());
System.out.println(s.nextDouble());
System.out.println(s.next());
```

Will print:

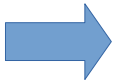
```
10
15.0
hello
```

Scanner

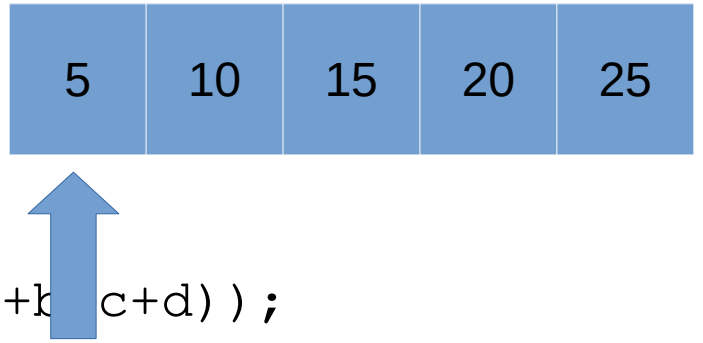
- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.



```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```

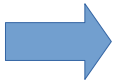


Scanner

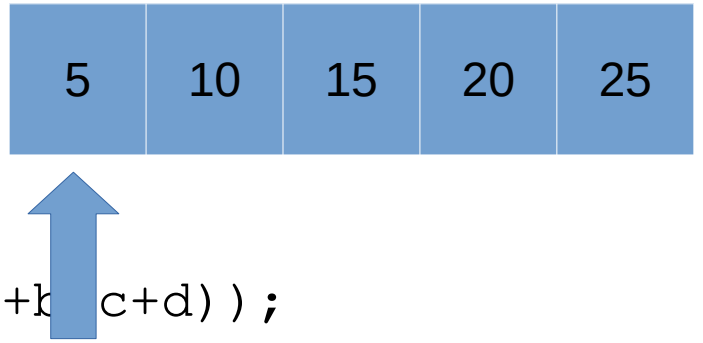
- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.



```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```



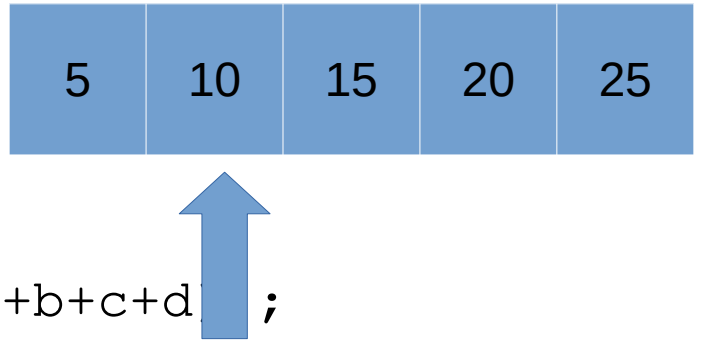
Scanner

- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```

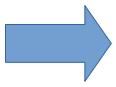


Scanner

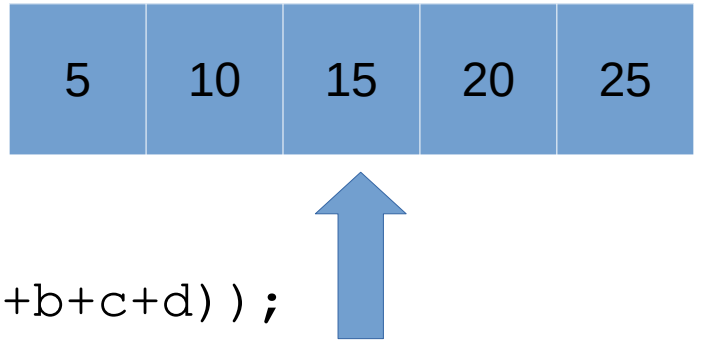
- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.



```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```



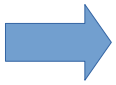
Scanner

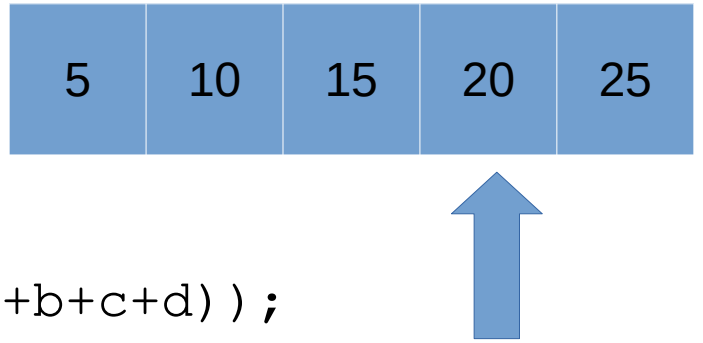
- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```





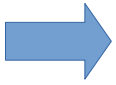
Scanner

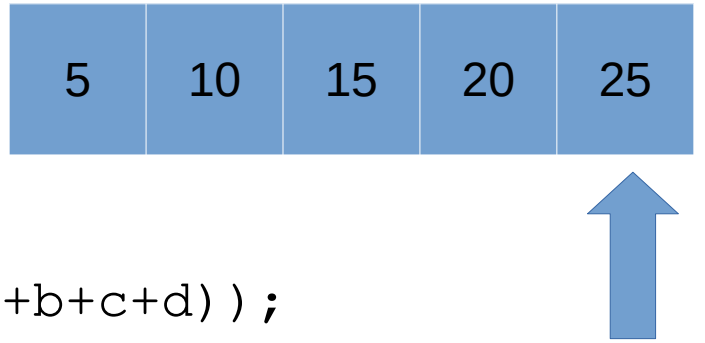
- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```





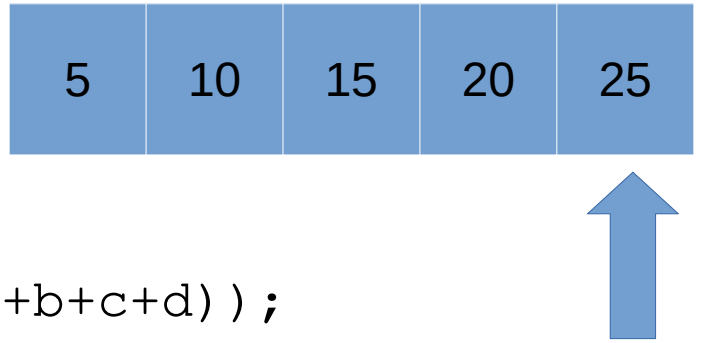
Scanner

- Lets us read structured data from *standard input*.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
String answer = String.format("%d + %d = %d", a, b, a+b);
```

- `Scanner` *consumes* its input with each call to `nextInt`, `nextDouble`, etc.

```
Scanner s = new Scanner(System.in);
int a = s.nextInt();
int b = s.nextInt();
int c = s.nextInt();
int d = s.nextInt();
System.out.println("a+b+c+d = " + (a+b+c+d));
```




```
a+b+c+d = 50
```

Scanner

- Lets us “look before we jump”

```
Scanner s = new Scanner(System.in);
if(!s.hasNextDouble()) {
    System.out.println("You have to enter a number!");
} else {
    double value = s.nextDouble();
    String result = String.format("%f^2 = %f", value, value*value);
    System.out.println(result);
}
```

- Will throw a `NoSuchElementException` if we try to read something that’s not there

```
Exception java.util.NoSuchElementException
        at Scanner.throwFor (Scanner.java:937)
        at Scanner.next (Scanner.java:1594)
        at Scanner.nextDouble (Scanner.java:2564)
```

Good luck, and Merry Christmas!