Type Systems

As a programmer, you need a basic understanding of type systems.

A **type system** is a collection of rules that assign a property called a **type** to the various constructs (variables, expressions, methods etc.) that comprise a computer program.

The main goals of type systems are:

1. Reduce bugs
2. Make code more readable
3. Enable compiler optimisations

We'll explore these in later slides

Different languages take very different approaches to types.

We'll mainly consider the Java and Ruby type systems.

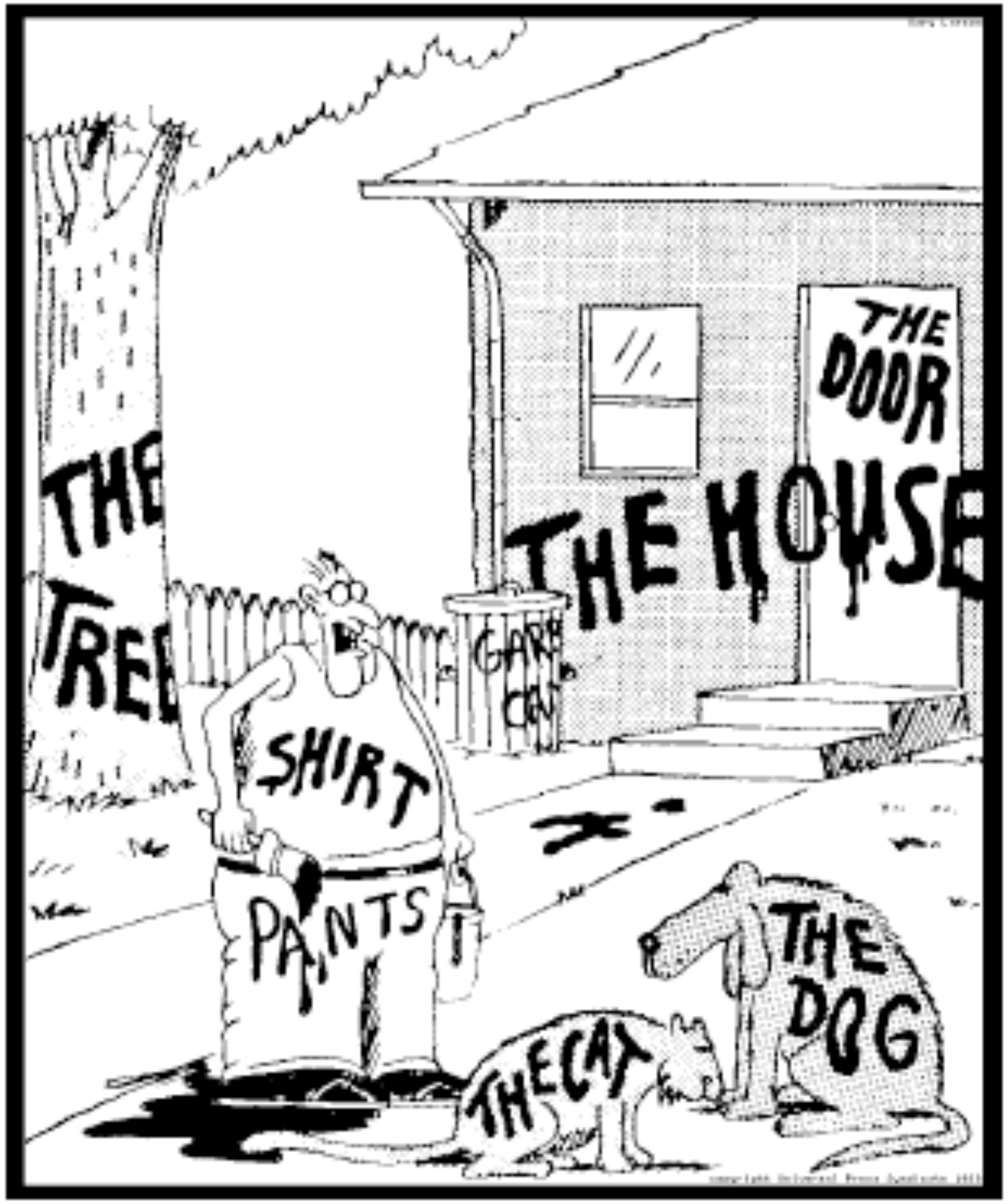
What is a **type**?

The type of something tells you what what you can do with it, for example:

Type	**Available Operations**
football	kick, head, sit on, spin on finger
house	live in, paint, buy, sell…
dentist	fill tooth, extract tooth, pay, …

So you know that you can spin a football on your finger, but you probably shouldn't do that to a dentist or a house.

Gary Larson on Static Types



"Now! *That* should clear up a few things around here!"

Types in Programming Languages

When you give a variable a **type**, you are saying how is should be used, e.g. (Java):

```
int count;
...
count = 5;
..
if (count == 20){
}
```

count is of type integer

assign count an integer, ok!

compare integer to integer, ok!

If you try to misuse the variable, the compiler complains:

```
if (count == “fifteen”){
}
```

compare integer to string, type mis-match error!

Types help prevent bugs

When you give a variable a type, the system can check that it is subsequently used correctly in accordance with its type.

This means that type errors are prevented — so a whole range of bugs are avoided.

Types make code more readable

In the above example, by writing (in Java)

```
int count;
```

we made it clear to anyone reading the code what range of values **`count`** can take on what what we can do with **`count`**.

By contrast, Ruby doesn't associate an identifiers with a type, so while you might initialise **`count`** thus:

```
count = 10
```

you could later assign **`count`** a string:

```
count = 'Dracula'
```

Legal, but not good as it makes the code hard to read.

Strong and Weak Typing

Type systems are sometimes classified as **strong** or **weak**. (These are relative terms, not absolute ones).

A strong type system prevents type errors from occurring.

A weak type system allows the possibility of type errors.

It may seem that strong type systems are better, but a very strong type system can be obstructive!

Example of weak typing on the following slide.

strong/weak UNRELATED to static/dynamic!

What does weak typing look like?

Weak typing is not in vogue; the original C language is the best-known weakly-typed language which could lead to highly complicated run-time errors.

Here's a sample of what can be done in C:

```
int x = 5;
```

x is an integer initialised to 5

```
char y[] = "37";
```

y is a string initialised to "37"

```
char* z = x + y;
```

z points to the string located 5 characters after y

The '+' in this example adds an integer to a pointer. This seems like it should be a type error, but a weak type system trusts that the programmer knows what they are doing.

Static and Dynamic Typing

Type systems are often classified as **static** or **dynamic**.

In **static typing**, type checking is performed at **compile time**.

In **dynamic typing**, type checking happens at **runtime**.

Don't confuse strong/weak typing with static/dynamic typing.

Static vs. Dynamic Typing

Static typing aims to catch as many errors as possible at **compile-time.** Traditional wisdom is that this is required for serious, production development (C++, C#, Java, …).

Dynamic typing is better for quick development, but more error-prone. Traditionally seen as more suitable for rapid prototyping or non-production development.

The current trend in language is in favour of dynamically-typed languages such as Ruby, Python, Scala etc.

Static Typing

In static typing, you tell the compiler the types of your variables, arguments etc. and it stops you accidentally misusing them.

This is a good safety mechanism, and allows the compiler to catch a whole range of bugs (type errors).

Static typing is heavily used in many production languages, e.g., C++, Java, C# etc.

However, it has its limitations...

Ruby Example (1)

Here we look an example of a run-time type error that can occur in a language that uses dynamic typing, but could not occur when using static typing.

This method takes a name and searches the **`persons`** array for a person of the name, returning false if the search fails.

```
def find_person name
  @persons.each do |person|
    if (person.name == name)
      return person
    end
  return false
end
```

What type of object does this method return?

If a programmer assumes that **`find_person`** returns a **`person`**, an error is likely to occur if it returns **`false`**…

Ruby Example (2)

Now say I use that method to find the Person object for "John Newman", who **isn't** stored in this structure:

```
person = people.find_person('John Newman')
```

And later on we process this person object, e.g.:

```
  puts person.name
```

This will report the following error on execution:

'NoMethodError: undefined method 'name' for false:FalseClass'

Doing this example in Java

Try to write the **`find_person`** method in Java.

```
public <return_type> find_person(string name){
    ...
}
```

You simply can't. The Java type system prevents you from mimicking the Ruby example.

The method must have a return type; it can't return a **`Person`** object sometimes and a **`Boolean`** other times.

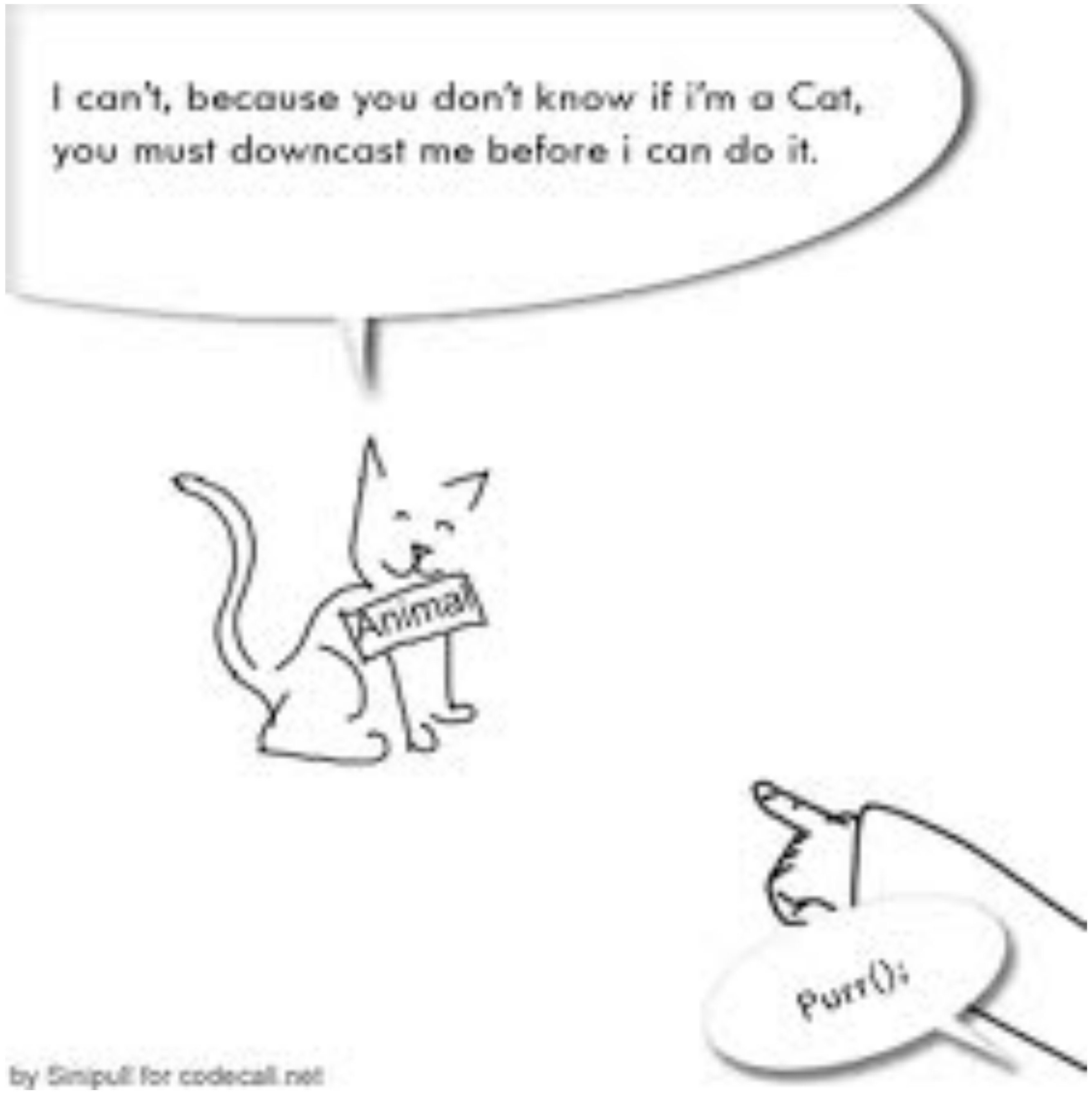
So the Java type system prevents the kind of bug from occurring.

Static typing isn’t always good!

It might seem that static typing is a good thing, but this isn’t necessarily the case.

Now we look at an example where static typing stops us from doing something that know would be fine!

Static typing can get in the way


I can't, because you don't know if i'm a Cat, you must downcast me before i can do it.
Animal
Purr();
by Sinipull for codecall.net

Cat example in Java

This creates a list of Animal objects and stores a Cat in the first place on the list (assume Cat is a subclass of Animal):

```
List<Animal> animals = new ArrayList<Animal>;
animals.add(new Cat(“cleo”));
```

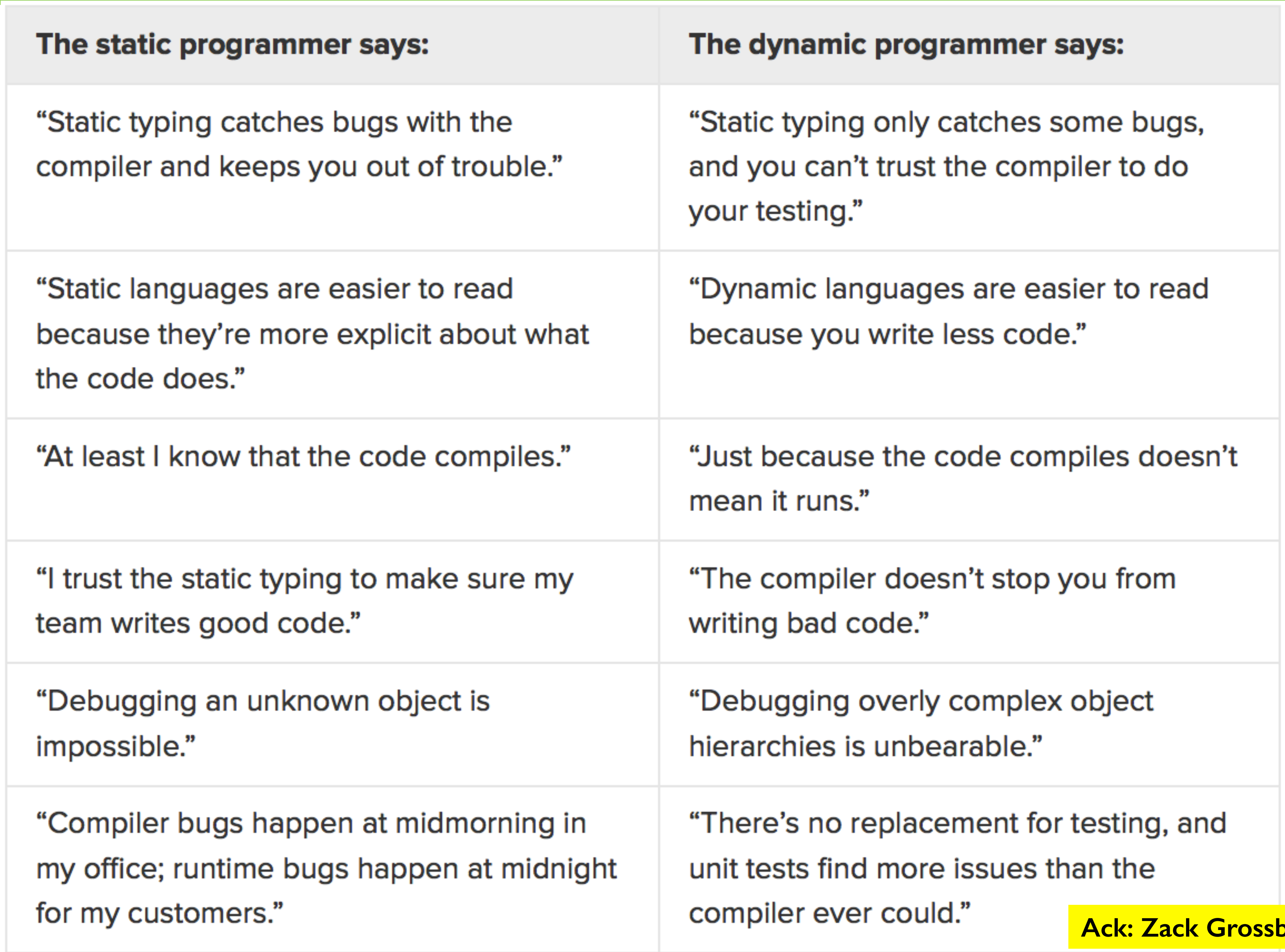
Later on, you want to write this code:

```
my_cat = animals.get(0);
my_cat.meeow();
```

You know it should work because you know the first element on the list is really a cat, but the compiler will report something like:

```
The method meeow() is not available for Animals.
```

Static vs. Dynamic Typing: a programmer's perspective



The static programmer says:	The dynamic programmer says:
"Static typing catches bugs with the compiler and keeps you out of trouble."	"Static typing only catches some bugs, and you can't trust the compiler to do your testing."
"Static languages are easier to read because they're more explicit about what the code does."	"Dynamic languages are easier to read because you write less code."
"At least I know that the code compiles."	"Just because the code compiles doesn't mean it runs."
"I trust the static typing to make sure my team writes good code."	"The compiler doesn't stop you from writing bad code."
"Debugging an unknown object is impossible."	"Debugging overly complex object hierarchies is unbearable."
"Compiler bugs happen at midmorning in my office; runtime bugs happen at midnight for my customers."	"There's no replacement for testing, and unit tests find more issues than the compiler ever could."

Ack: Zack Grossbart

Classes and Types

It’s easy to confuse **class** and **type** in an object-oriented language.

The **type** of an object is what you can do with it, i.e. its public methods.

The **class** of an object is its type (public methods) plus the details of how the methods are implemented:

class = type + implementation

References and Objects

To add to the mix, a reference and the object it refers to may be of different types.

Say **`Customer`** is a subclass of **`Person`**, then in Java we can write:

```
Person my_person = new Customer(…);
```

Here the reference **`my_person`** is of type **`Person`**, while the object it refers to is of type **`Customer`**.

Java Typing in a Nutshell

Java is regarded as **strongly-typed**. Objects and object references are given types and these are checked to prevent type errors.

Run-time type errors can occur (e.g., ClassCastException) so it's not completely type-secure.

Java primarily uses **static** typing. The compiler checks that object references are type safe.

The runtime system also check object types where necessary, so some dynamic typing occurs as well.

In Java, as in C++, dynamic typing is inextricably linked with the inheritance relationship.

The Ruby Type System

Ruby is also regarded as **strongly-typed.** All objects have a type and whenever an object is sent a message, the runtime checks that the object can respond to it.

However, **variables have no type** in Ruby. Any variable can refer to any object.

In this sense, Ruby variables are **typeless**.

Ruby is interpreted so it cannot use **static** typing.

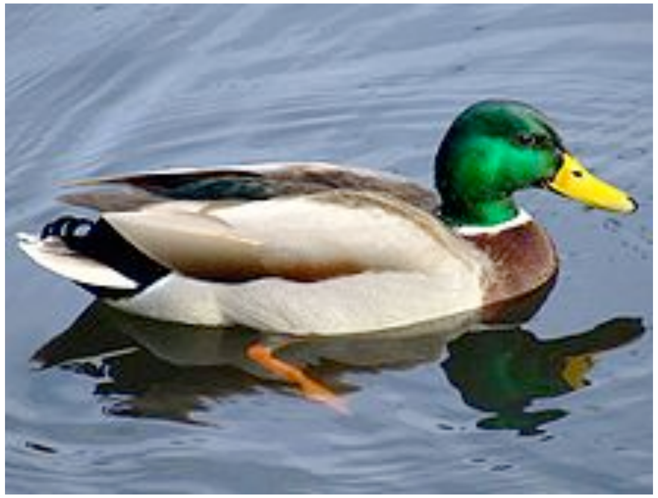
Ruby is entirely **dynamically typed**.

In fact, it uses a simple system called **duck typing**.

Duck Typing

Ruby typing is commonly called **Duck Typing**.

Types are not checked until the code is executed. You ask an object to do something. If it can it does, if it can't, it throws an exception.



"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." -- James Whitcomb Riley

In this model, the class of an object does not matter, all that matters is what it can do.

What if the duck doesn’t quack?

What happens if we write this in Ruby?

```
person = Person.new(…)
person.quack
```

We’ve seen how method lookup works before; here the runtime system will not find a **`quack`** method in the Person class or any of its superclasses.

It then sends the **`person`** object the **`method_missing`** message. You can implement **`method_missing`** to handle this situation.

If **`method_missing`** isn’t implemented either, it throws a **`NoMethodError`** exception.

Type Systems Summary

The **type system** of a programming language is a very important feature of the language.

Types help to reduce bugs, make code clearer and help the compiler perform certain optimisations.

Type systems can be strong or weak, static or dynamic (or a hybrid of these).

Ruby is strongly typed, and uses a form of dynamic typing called **duck typing**.

This is important to understand as we look further at **polymorphism**.