<u>program and compiler</u>
a compiler is a software tool that reads a program written in a higher level programming lang and translates it into machine code

an interpreter reads the entire prog and interprets what each statement means and executes it directly

CT errors always better than RT errors

<u>variables and types</u>
variables are abstractions for a piece of data in memory
- referring to memory slots are not user friendly
- location of variable may change

purpose of types
- what data type the variable is an abstraction over - communicated to readers
- what operations are valid on this variable - communicated to the compiler/interpreter
- how this operation behaves - communicated to the compiler

dynamic vs statically typed

dynamic	same var can hold values of unrelated types **type is attached to value**
static	once a var is declared, its type is immutable but value is mutable **type is attached to var**

strong typing vs weak typing

strong	strict rules in type system to ensure type safety - catches type errors during compile time
weak	more permissive in terms of type checking

primitive types in Java

boolean	boolean	1-bit
character	char	16-bit
integral	byte short int long	8 - bit 16 - bit 32 - bit 64 - bit
floating point	float double	32-bit 64-bit

subtypes
T <: S to denote that T is a subtype of S
the subtyping relationship in general must satisfy two properties:
- reflexive: for any type S, we have S <: S (S is a subtype of itself)
- transitive: if S <: T and T <: U, then S <: U

byte <: short <: int <: long <: float <: double
char <: int

Java allows a variable of type **T** to hold a value from a variable of type **S** only if **S <: T**, this is a **widening type conversion**

Java does not allow explicit narrowing type conversions because it can potentially lead to loss of data or precision
- can only be done through type casting

<u>functions</u>
functions are an abstraction over computation
black-boxing

abstraction barrier
- above the barrier, the concern is about using the function to perform a task
- below the barrier is the concern on how to perform said task

<u>encapsulation</u>
a **composite data type** allows programmers to group primitive types together, give it a name to become a new type, and refer to it later

<u>classes</u>
- bundling of composite data types and its associated function on the same side of the abstraction barrier together

<u>OOP</u>
- to model a problem in an object-oriented manner, we typically model nouns as classes and objects
- properties or relationships among classes as fields
- verbs or actions of corresponding objects as methods

<u>reference types</u>
- reference variables store only the reference to the value and therefore, two reference variables can share the same value
- any reference variable that is not initialized will have the special reference value **null**

<u>info hiding</u>
this keyword
- **this** is a reference variable that refers back to the calling object itself
- the **this** reference makes it explicit that the expression is referring to a field in the class, rather than a local variable or a parameter

<u>tell, don't ask</u>
instead of getting the vars from the class and performing a method on it, do it within the method itself

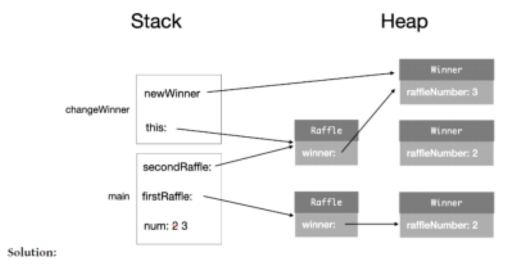
<u>class methods</u>
similar to a static field, a static method is associated with a class, not an instance of the class
- the reference **this** has no meaning within a class method with the static method and will throw an error when compiled

<u>composition vs inheritance</u>
- **Inheritance** establishes an "is-a" relationship between classes. For example, if class B inherits from class A, it means that B is a type of A
- **Composition** establishes a "has-a" relationship between classes. For example, if class A contains an object of class B, it means that A has a B

@qilstiano on GitHub

<u>stack and heap diagram</u>
example:
```
class Winner {
        private int raffleNumber;
        public Winner(int raffleNumber) {
                this.raffleNumber = raffleNumber;
        }
}

class Raffle {
        private Winner winner;
        public Raffle(Winner winner) {
                this.winner = winner;
        }
        public void changeWinner(Winner newWinner) {
                this.winner = newWinner; // Line A
        }
}
Main
int num = 2;
Raffle firstRaffle = new Raffle(new Winner(num));
Raffle secondRaffle = new Raffle(new Winner(2));
num += 1;
secondRaffle.changeWinner(new Winner(num));
```



<u>compile/run-time types & dynamic binding</u>

```
class Animal {}
class Dog extends Animal {}
public class Main {
    Animal animal1 = new Animal(); // Compile-time & Runtime type: Animal
    Animal animal2 = new Dog();    // Compile-time: Animal, Runtime: Dog
```

1. **COMPILE TIME**
- find method signature (name + parameters) in compile time type of calling object
- store method descriptor in the compile code - cannot change anything after this
- will go for most specific match

2. **RUN TIME**
- try to find the same descriptor in run time type of calling object, exact match of the descriptor (i.e. the parameter does not matter)

class methods do not support dynamic binding
- method to invoke is resolved statically during compile time

<u>methods overloading and overriding</u>
- **method signature of a method**
  - method name
  - number of parameters
  - type of parameters
  - order of parameters
- **method descriptor**
  - method signature + return type
- **method overriding**
  - used to alter the behaviour of an existing class
  - @Override
  - use super to access overridden methods (e.g. super below refers to default toString() method

```
@Override
public String toString() {
        return super.toString() + " . . .";
}
```

check that the **method signature** of a method that has the tag `@Override` is actually the same as that of the method that is being overridden
when a method is overridden - the overridden method should throw the same exception or any subtype of the exception
R r = a.foo();

R must return r or its sub types - super types is not ok

Liskov Substitution Principle

a subclass should not break the expectations set by the superclass.

- Returning an object that's incompatible with the object returned by the superclass method.
- Throwing a new exception that's not thrown by the superclass method.
- Changing the semantics or introducing side effects that are not part of the superclass's contract.

example:
method: class.method() returns 'A' 'B' or 'C'
another method: display() takes ^ as input

if we inherit from class and override the method():
    now it outputs 'S' and 'U'
LSP is broken because display is now not working

final

we have now seen that the final keyword can be used in three places:

- in a class declaration to prevent inheritance.
- in a method declaration to prevent overriding.
- in a field declaration to prevent re-assignment.

abstract classes

an abstract class in Java is a class that has been made into something so general that it cannot and should not be instantiated (a contract)

An abstract method cannot be implemented and therefore should not have any method body.

```
interface GetAreable {
  public abstract double
getArea();
}
```

interfaces as supertypes

- if a class C implements an interface I, C <: I
- this implies that a type can have multiple supertypes

casting using an interface

- like any type in Java it is possible to cast a variable to an interface type even if the variable's class does not implement the interface
- this works because the compiler trusts that the programmer knows what they are doing because there is a possibility that a subclass could implement the interface

wrapper classes

a wrapper class

- a class that encapsulates a type, rather than fields and methods
- can be used just like every other class in Java and behaves just like every other class in Java
- they are reference types, their instances can be created with new and stored on the heap

why? through polymorphism and overriding of methods, wrapper classes help work on primitive types

wrapper classes (continued)

auto-boxing and unboxing

- conversion back and forth between a primitive type and its wrapper class is provided as a feature by Java

```
Integer i = 4; //auto-
boxing of primitive value
of 4 is converted to
Integer
int j = i; //auto-unboxed
wrapper class Integer to
int
```

performance

- since primitive wrapper class objects are immutable, every time a wrapper object is updated, a new wrapper object is created
    - due to autoboxing and unboxing, the cost of creating objects is hidden and forgotten by users

variance

variance of types

- subtype relationship between classes and interfaces are based on inheritance and implementation

variance of complex types

let's let C(S) correspond to some complex type based on type S (an array of type S is an example of a complex type)

Variance	Condition
Covariance	if S <: T implies C(S) <: C(T)
Contravariant	if S <: T implies C(T) <: C(S)
Invariant	neither covariant not contravariant

this however, opens up the possibility of run-time errors, even without typecasting.

- e.g. a String is able to be entered into an Object[]

exceptions and warnings

`try` block

- main task for the code

`catch` block

- error handling comes under the `catch` clause, each handling a different type of exception
- exceptions are instances that are a subtype of the `Exception` superclass

throwing exceptions

- declare that the constructor/method is throwing an exception with the throws keyword
- create a new `Exception` object and throw it to the caller with the throw keyword

note that executing the throw statement causes the method to immediately return and subsequent code blocks do not execute

creating our own exceptions

- an exception can be created by simply inheriting from one of the existing ones

```
class
IllegalCircleException
extends
IllegalArgumentExceptio
n { … }
```

when a method is overridden - the overridden method should throw the same exception or any subtype of the exception

generics

a generic type takes other types as type parameters
generics are **invariant** in Java

generic methods
methods that have been parameterized with a type param
`public static <T> boolean contains(T[] array, T obj)`

bounded type parameters
`public <T extends GetAreable> T find(T[] array)`
T must be <: GetAreable

type erasure

- At runtime, generic types are treated as their raw types (non-generic).
    class A<T extends Comparable<T>> { method(T t) }
    - at RT, T is erased to Comparable
- Arrays retain their type information at runtime, while generics lose it due to type erasure.
- Reifiable Type: A type whose type information is fully available at runtime. Non-generic types, arrays, and raw types are reifiable.
- Heap Pollution: Heap pollution occurs when a collection containing raw types is assigned to a collection with a parameterized type, causing type safety issues.

raw types

- a raw type is a generic type used without type arguments
- example:
```
Seq<String> s = new Seq<String>(4);
s.set(0,1234); //compiler throws
unchecked warning but will allow it
```
- without a type argument, the compiler can't do any type checking - since we are using the raw type of Seq here
- ClassCastException if we String a = s.get(0);

wildcards

- upper-bounded wildcards
    - an upper-bounded wildcard is an example of covariance
    - the upper-bounded wildcard has the following subtyping relations:
        - if S <: T, then A<? extends S> <: A<? extends T> (covariance)
        - for any type S, A<S> <: A<? extends S>
-
- lower-bounded wildcards
    - an example of contravariance
        - if S <: T, then A<? super T> <: A<? super S> (contravariance)
        - for any type S, A<S> <: A<? super S>

type inference

rules for type inference

1. figure out all of the type constraints on our type parameters, and then solve these constraints
2. if no type can satisfy all the constraints, we know that Java will fail to compile

when resolving the type constraints for a given type parameter T we are left with:

- `Type1 <: T <: Type2`, then T is inferred as `Type1`
- `Type1 <: T`*, then T is inferred as `Type1`
- `T <: Type2`, then T is inferred as `Type2`

`Type1` and `Type2` are arbitrary types

*: Note that `T <: Object` is implicit here. We can see that this case could also be written as `Type1 <: T <: Object`, and would therefore also be explained by the previous case (`Type1 <: T <: Type2`).

@qilstiano on GitHub