CS2030 Overview: Generics

Compilation Time vs Run Time
Compilation:
Compilation time is when the JVM compiles the code into byte code.
As such the compiler does not know what value a variable will take on, nor the sequence of execution of the program.

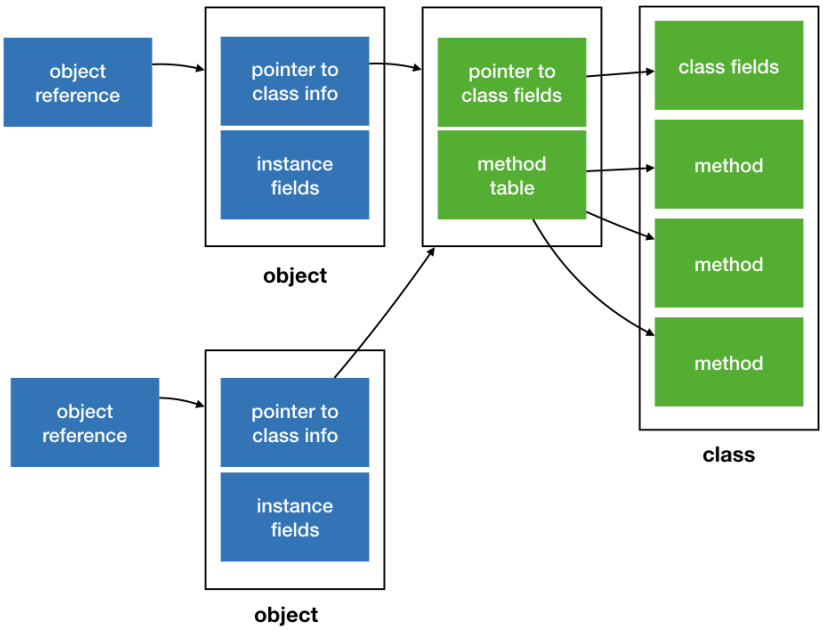
Hence the compiler is conservative and adopts late binding.

Late Binding:

In OO languages, you can have methods named print( ) implemented differently in different classes.

```
Printable[] objs;
:
// initialize array objs
:
for (Printable o: objs) {
o.print();
}
```

The compiler cannot always be sure what is the class of the object the variable o will refer to during runtime, and thus, it cannot be sure which version of print method will be executed. bindings of print() to the actual set of instructions will only be done at run time, after the object o is instantiated from a class.



When o.print() is invoked, Java refers to the method table, which points to either the method table for Circle or for Point, based on the class the object is an instance of.

This behaviour, which is common to OO programming languages, is known as polymorphism, and allows overloading of methods in subclasses. (methods with the same name but different input parameter).

SubTyping in Java:

Subtyping in Java runs as such:
`byte` <: `short` <:`int` <: `long` <: `float` <: `double`; and `char` <: `int`

In runtime, java interpretes the types and perform implicitly:

Type Widening:
When a small primitive type value is automatically accommodated in a bigger/wider primitive data type, this is called widening of the variable.
Eg:
Int I = 10;
Long j = I;

This runs with no dataloss or error.

Type Narrowing:

However, the reverse process does not happen implicitly, a primitive variable must be manually typecasted down, and more often than not, the typecast will result in dataloss.
Eg:
int i=198;
byte j=(byte)i;

A similar concept is somewhat applied when we Introduce Generics.

Generics:
Generics, a way to make wrapper classes that can work with any kind of non-primitive datatype as long as it adheres to the responsibilities stated within the class.

Generics are Invariant with respect to a type parameter:
If a class or interface of type B is a subtype of A, then neither is C<B> a subtype of C<A>, nor is C<A> a subtype of C<B>.
Queue<Circle> qc = new Queue<Shape>(); // compile-time error
Queue<Shape> qs = new Queue<Circle>// compile-time error
Queue<Shape> qs = (Queue<Shape>) new Queue<Circle>(); // compile-time error
Attempting to type cast just like when we do a narrowing reference conversion will fail as well.

In contrast, if a class or interface B is a subtype of A, then B<T>; is Also a subtype of A<T> (they are covariant).

Subtyping amongst generic types, ie if T is a subtype of S, we can denote this by using <?>

<?> is not a type itself, but a notation to denote variance of generic types.

Bounding wildcards
Java uses the keyword “extends" and “super” to specify the bound
If T is a subtype of S, then Queue<T> is a subtype of Queue<? extends S>.

For instance:
Queue<Circle> and Queue<Shape> are both subtypes of Queue<? extends Shape>.

Queue<Object> is not a subtype of Queue<? extends Shape>

Queue<? extends T> is a subtype of Queue<? extends S>.the use of ? extends is covariant.

If T is a subtype of S, then Queue<S> is a subtype of Queue<? super T>:

Queue<Shape> and Queue<Object> are both subtypes of Queue<? super Shape>.

Queue<Circle> is not a subtype of Queue<? super Shape>

The use of <? super > is contravariant.

We can replace ? extends X with type X or any subtype that extends X; <? super X> with type X or any **supertype** of X.

Type Erasure:

In Java, for backward compatibility, generic typing is implemented through a process called type erasure.

type argument is erased during compile time, and the type parameter T is replaced with either **Object** (if it is unbounded) or the bound (if it is bounded).
For instance:
Queue<Circle> will be replaced by Queue and T will be replaced by Object

Queue<? extends Shape> will be replaced with Queue and T will be replaced with Shape

The compiler also inserts type casting and additional methods to preserve the semantics of generic type.

Java's design decision to use type erasure to implement generics has several important implications.

Java does not support generics of a primitive type. We cannot create a Queue<int>, for instance. We can only use reference types as type argument (anything that is a subtype of Object).

Cannot have code that looks like the following:

```
class A {
  void foo(Queue<Circle> c) {}
  void foo(Queue<Point> c) {}
}
```

Even though both foo above seem like they have different method signatures and is a valid application of method overloading, due to type erasure the compiler translate them both to

```
class A {
  void foo(Queue c) {}
  void foo(Queue c) {}  // compile-time error
}
```

using static methods or static fields in generic classes becomes trickier

```
class Queue<T> {
  static int x = 1; // no compile-time error, but dangerous! Only one copy of x even for
different                                    // type-parameterised queues
  static T y; // compile-time error
  static T foo(T t) {}; // compile-time error
}
```

Queue<Circle> and Queue<Point> both gets compiled to Queue. So they share the same x, even though you might think Queue<Circle> and Queue<Point> as two different distinct classes.

Also, there will only be a copy of y shared by both Queue<Circle> and Queue<Point>, the compiler does not know whether to replace T with Circle or Point, and will not compile. cannot create an array of parameterised types.

```
Queue<Circle>[] twoQs = new Queue<Circle>[2]; // compiler error
twoQs[0] = new Queue<Circle>();

twoQs[1] = new Queue<Point>();
(After type erasure…)
Queue[] twoQs = new Queue[2]; // compiler error
twoQs[0] = new Queue();
twoQs[1] = new Queue();
```

If Java would have allowed us to create an array of a parameterised type, then we could have added an element of incompatible type Queue<Point> into an array of Queue<Circle> without raising a run-time error.

Generics in Java was added relatively recently and to maintain backward compatibility, the compiler has to erase the type parameter during compile time. During runtime, the type information is no longer available, and this design causes the quirks above.

Note: for backward compatibility, Java allows us to use a generic class to be used without the type argument. For instance, even though we declare Queue<T>, we can just use Queue.
Queue<Circle> cq = new Queue();

This is called a raw type. Recent Java compilers will warn you if you use a raw type in your code.

We can get around the problem of not being able to create generics of primitive types using wrapper classes.

Java provides a set of wrapper classes, one for each primitive type:
Boolean, Byte, Character, Integer, Double, Long, Float, and Short.

Java 5 introduces something called autoboxing and unboxing, which creates the wrapper objects automatically (autoboxing) and retrieves its value (unboxing) automatically.

Using an object comes with the cost of allocating memory for the object and collecting of garbage afterward, it is less efficient than primitive types.

All primitive wrapper class objects are immutable -- once you create an object, it cannot be changed. Thus, every time sum in the example above is updated, a new object gets created.

Autoboxing and unboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.
Does not work for conversion between wrapper objects!
Eg:
Double a = 1.0; // ok
Integer b = 1; // ok

Integer a = new Integer(1); // ok
Integer b = new Integer(1.0); // compiler error
Double c = new Double(1.0); // ok
Double d = new Double(1); // ok

Double a = 1; // compiler error
Integer b = 1.0; // compiler error
Integer c = (int) 1.0; // ok

Integer a = new Integer(1);
Double b = a; // compiler error
Double d = (int) a; // compiler error
Double c = (double) a; // ok

```
double a = (Integer) 1; // ok
int b = (Integer) 2; // ok

double d = (Double) 5; // compiler error
int i = (Integer) 2; // ok
```

We can get around the problem of not being able to define static methods using generic methods.
Generic methods are just like generic type, but the scope of the type parameter is limited to only the method itself.

```
class Queue<T> {
  static <T> T foo(T t) { return t; };
}
```

The above would compile. Note that the <T> in static <T> T foo(T t) effectively declares a new type parameter T, scoped to the method foo, and has nothing to do with the type parameter <T> for Queue<T>.

As a convention, it is common to use the single letter T here, which may be confusing. We can actually use any other single letter to represent the type parameter.

```
class Queue<T> {
  static <X> X foo(X t) {
  return t;
  };
}
```

To invoke a generic method, we can call it like this Queue<Point>foo(new Point(0, 0)); or
Queue.foo(new Point(0, 0));

Type inference:
In the last example, Java compiler uses type inference to determine what T should be: in this case, we are passing in a Point object, so T should be replaced with Point.

Type inference can also be used with constructors:

```
Queue<Point> q = new Queue<>();
```

In the line above, we use the diamond operator <> to implicitly ask Java to fill in the type for us. Again, type inference allows Java to infer that we are creating a Point object.

In cases where the type inference finds multiple matching types, the most specific type is chosen.

Overall:
Java Generics are a useful tool to make it more convenient to use wrapper classes to code in a functional paradigm.