

Bantam Java Language Manual



Marc L. Corliss

Hobart and William Smith Colleges

corliss@hws.edu

<http://math.hws.edu/mcorliss>

E Christopher Lewis

VMware

lewis@vmware.com

<http://www.eclewis.net/>

©2007, Marc L. Corliss and E Christopher Lewis

This manual is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 2.5 License (<http://creativecommons.org/licenses/by-nc-nd/2.5/>). This license allows you to duplicate and distribute this manual in unmodified form for non-commercial purposes. See the license for more details.

Note: this document was extracted from a larger document covering all aspects of the Bantam Java compiler project. It is available at <http://www.bantamjava.com>.

Contents

1 Overview	1
2 Inheritance	3
3 Class Definitions	4
4 Member Definitions	5
5 Statements	11
6 Expressions	17
7 Built-In Classes	29
8 Bantam vs. Java	32
References	35

This manual presents the Bantam Java programming language, the source language for the compiler. Bantam Java is a subset of the Java language. This manual discusses those Java features that were included and excluded from the Bantam language. It also describes in some detail the included features, although for a more detailed explanation one should consult a Java reference [1]. This manual is intended for students who are already familiar with the Java programming language and its features.

1 Overview

Bantam Java (or Bantam, for short), like Java, is an object-oriented programming language. As described in numerous textbooks [2, 1] (too many to cite all of them), an object is a self-sufficient component that has some data and behavior associated with it, which is meant for modeling a real world entity such as a player, animal, room, *etc.* To construct an object in Bantam or Java, one defines a class, which is effectively the blue print for all objects generated from that class. In other words, an object is an *instance* of a class. The class defines both the data elements and behavior of any object constructed from that class.

A class in Bantam Java and Java is essentially a *type*. A type specifies how a value (in this case an object) of that type can be used. Bantam Java, like Java, is a *strongly-typed language*, meaning that the compiler, either during compilation or at runtime, prevents a value of one type being used where a value from a different type is required (there is an exception to this rule, which will be discussed in the next section). In other words, the compiler effectively prevents values from being used in an improper way.

A Bantam program consists of one or more files (with suffix “.btm”), each of which contains one or more class definitions. All code must reside within a class definition in Bantam. For this reason, classes and objects are critical and necessary components of any Bantam program.

Each class in Bantam, contains zero or more *members*. A member is either a *field* (part of the object’s data) or a *method* (part of the object’s behavior). A field is one kind of *variable*, which can hold various data values throughout the course of the owner object’s lifetime. A field has a type, which specifies the type of data it can store. A method, on the other hand, is effectively a name for a block of code, which can be executed on behalf of the owner object. The method can be executed by referring to its name, which is known as a method *call*. We will show later how fields and methods can be declared and used.

Unlike in Java, all fields and methods must be instance members, that is to say they cannot be declared **static**. In other words, each member is specific to a particular object; there are no class-wide members in Bantam.

When an object is created, the syntax and semantics of which are discussed later, it is allocated its own copy of each field. The object's field values may differ from the field values of other objects created from the same class. When an object's method is called, it may use the object's fields, therefore the outcome of a method depends largely on the owner object. Before an object is created, it is set to the special value **null**, which loosely speaking means it is an empty object.

One of the classes defined in a Bantam program must be called **Main**. This class must contain a method called **main** (among possibly other members). When a Bantam program is started, a **Main** object is created and the **main** method is called. The program terminates when the **main** method finishes. The **main** method does all of the work of the program or calls methods that do the work on its behalf. Note that this starting point is different than in Java. In Java, which has static members, a static **main** method within a user-specified class is initially called.

There are also some built-in classes that allow the programmer to perform input and output to the screen or to a file (**TextIO**), perform system related tasks such as exiting the program (**Sys**), and store and manipulate sequences of characters (**String**). These are discussed in Section 7.

In addition to classes, there are two primitive types of data in Bantam: **int** and **boolean**. A primitive, unlike an object, holds a simple value. An **int** holds a 32-bit signed integer. A **boolean** holds either **true** or **false**. Other primitives from Java, such as **double** and **long** were not included in Bantam. These additional types add little intellectual challenge to the design and implementation of the compiler and for this reason were excluded from the language. As in Java, primitive types can be used in similar places as object types (*e.g.*, a field can have type **int**).

One important distinction to make is between primitive variables and object variables. A primitive variable (*e.g.*, a field) holds the value (*e.g.*, 42). However, an object variable does not hold an object itself, but rather a *reference*, or more precisely a memory pointer, to the object. If two or more object variables refer to the same object, then we say that they *alias*.

Memory management in Bantam Java is similar to Java. Memory allocation occurs when objects are constructed (using **new**). Memory deallocation is handled implicitly, *i.e.*, a garbage collector is responsible for deallocating memory. Unlike in Java, the garbage collector is disabled by default and can be enabled using the compiler flag “-gc”.

There are two types of comments in Bantam Java. The first is a single line comment, which

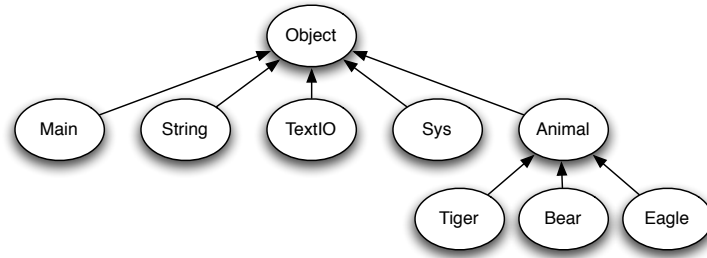


Figure 1: Example class hierarchy tree in Bantam.

begins with `//`. The text following `//` up to and including the end of the current line is ignored by the compiler (including `//`). The second type of comment is a multi-line comment that begins with `/*` and ends with `*/`. As the name suggests, these may span multiple lines. The compiler ignores all text in between `/*` and `*/` (including `/*` and `*/`). Multi-line comments may not be nested.

In the remainder of this manual, we will look in more detail at defining classes and class members. But before doing that, we discuss a core feature of Bantam and most object-oriented languages, namely *inheritance*.

2 Inheritance

As in Java, Bantam allows classes to extend other classes. If class **A** extends class **B** then class **A** inherits all of class **B**'s members including all the members that **B** inherits from other classes. We say that **A** is a child of **B** and **B** is the parent of **A**. If a class does not explicitly extend another class then it automatically extends the built-in class **Object**, which is similar to the **Object** class from Java (albeit with less functionality). Bantam and Java support only single inheritance: a class inherits from exactly one other class except the built-in **Object** class, which does not extend any other class.

To visualize the class dependencies, we can build a class hierarchy graph where each node in the graph represents a class. A directed edge is drawn from node **A** to node **B** if and only if class **B** is the parent of class **A**. Because Bantam only supports single inheritance, the class hierarchy graph of a well-defined Bantam program is actually a tree with the class **Object** as the root.

Figure 1 shows an example class hierarchy tree in Bantam. Notice in the figure that the class **Tiger** is a descendant of the class **Object** (it extends **Animal**, which extends **Object**). We say that **Tiger** is a *subclass* of **Object** and that **Object** is a *superclass* of **Tiger**. More precisely, a class **A** is a

subclass of a class **B** (or equivalently, **B** is a superclass of **A**) if we must pass through node **B** to get to the root of the class hierarchy tree from node **A**. Because a class is actually a type, we also say that **Tiger** is a *subtype* of **Object** and that **Object** is a *supertype* of **Tiger**. Each class is defined to be a subclass (or subtype) and a superclass (or supertype) of itself. The class **Object** is a superclass of all classes and is only a subclass of itself.

When a class **A** is a subclass of a class **B**, we also say that **A** *conforms* to **B** since **A** must contain all of the functionality of **B**. Therefore, anywhere that we can use an object of type **B**, we can also use an object of type **A**. The opposite is not true. Because **A** may have more functionality than **B**, an object of type **B** cannot be used in place of an object of type **A**.

One implication of conformance is that the type of an object variable at compile time may differ from the type at runtime. For example, we can construct a new **Tiger** object and store it in a variable of type **Object** since **Tiger** conforms to **Object**. The compile time type of the variable, which is often called the *static type*, is **Object**, while the runtime type, which is often called the *dynamic type*, is **Tiger**. The dynamic type is often unknown until the program is run.

We will have more to say about inheritance and conformance in our discussion of class members (4) and casting (6).

3 Class Definitions

The format of a class definition is:

```
class <name> [ extends <parent> ] {  
    <members>  
}
```

where **<name>** is replaced with the name of the class, **<parent>** is replaced with the name of the parent class, and **<members>** is replaced with the class members (fields and/or methods). Note: **[** and **]** are not part of the language, but are instead used to indicate that **extends <parent>** is optional. A class can explicitly extend another class by using **extends**. Alternatively, **extends** can be omitted, in which case the class automatically extends the built-in class **Object**.

Both **class** and **extends** are keywords in Bantam. Neither can be used as a name of a class, method, or variable. Bantam is case-sensitive, so **class** and **extends** are keywords, while **Class** and **EXTENDS** are not. The class name (**<name>**) and parent name (**<parent>**) are also case sensitive

(in fact, all tokens in Bantam are case sensitive). Therefore, a class named **Animal** is not the same as a class named **aNimal**. The names may contain any sequence of upper or lowercase letters, digits, or the symbol ‘_’, although the name must start with a letter. By convention (taken from Java), class names generally start with an uppercase letter. If a class name contains multiple words combined into one name the start of each new word begins with an uppercase letter. For example, if we wanted a class name to include both “Bantam” and “Java” we would name the class **BantamJava**.

Unlike in Java, class definitions cannot use modifiers such as **public** and **package**. Since there are no packages in Bantam these modifiers are not necessary and using one is illegal. In addition, Bantam does not include abstract and static classes so the **abstract** and **static** modifiers are also illegal. Bantam also does not include interfaces so a class cannot use the keyword **implements** nor can an interface be defined. Finally, classes cannot be nested within other classes.

To define a (trivial) class **Animal**, we write the following:

```
class Animal {  
}
```

This class implicitly extends **Object**. It defines no data elements and no behavior. To do this, we must define some class members.

4 Member Definitions

As stated above there are two kinds of class members: fields and methods. We look in turn at defining each, below.

Fields. The format for defining a field (*i.e.*, instance variable) is:

$$\langle \text{type} \rangle \langle \text{name} \rangle [= \langle \text{expression} \rangle] ;$$

where **<type>** is replaced with the field’s type, **<name>** is replaced with the field’s name, and **<expression>** is replaced with the field’s initialization expression. Note the ‘=’ and initialization expression are optional. The **<type>** must be either a primitive type (*i.e.*, **int** or **boolean**) or a class name (*e.g.*, **Object**). The **<name>** is a non-empty sequence of upper and lowercase letters, digits, and the special symbol ‘_’. Each field defined within a particular class must have a distinct name.

By convention, field names generally start with a lowercase letter. Like class names, when multiple words are combined into one variable name, the start of each word (except the first word) begins with an uppercase letter. For example, if we wanted a field name to contain “Bantam” and “Java” we would call it **bantamJava**. The beginning lowercase letter indicates it is not a class name and the subsequent uppercase letters indicate the separation of words embedded within the name. If the field is meant to be constant (*i.e.*, never changed) then the convention is to use all uppercase and use ‘_’ to indicate separation of words (*e.g.*, **BANTAM_JAVA**).

Below is another (slightly more interesting) definition of class **Animal**, which includes three field definitions:

```
class Animal {
    int numLegs = 4;
    boolean canFly;
    Animal mate;
}
```

The first field, **numLegs**, is a primitive variable with type **int**, which initially holds the value 4, although this can be changed later. The second field, **canFly**, is also a primitive variable, but with type **boolean**. It defines no initialization expression. When a field definition omits the initialization expression, it is automatically assigned a value based on its type. Fields with type **int** are automatically assigned 0, fields with type **boolean** are automatically assigned **false**, and fields with object types are automatically assigned **null**. Therefore, **canFly** is initially set to **false**. The third field is an object of type **Animal**. Its initial value is **null**, since it does not define an initialization expression. It has the same type as the class that is being defined, which is okay so long as we do not attempt to initially construct the object (object construction is discussed later in Section 6). If we try to construct a new **Animal**, we will end up in an endless cycle of **Animal** object constructions. This problem only occurs for objects with types that are a subclass of the defined class.

We could have also written the following equivalent code:

```
class Animal {
    int numLegs = 4;
    boolean canFly = false;
    Animal mate = null;
}
```

It would not be equivalent to omit the assignment of **numLegs** to the value **4** since **numLegs** would have been automatically assigned **0**.

The initialization expression can be arbitrarily complicated as discussed in 6. For example, we could have the following:

```
class Animal {
    int numLegs = 4;
    boolean canFly;
    Animal mate;
    int strength = (numLegs * 10) + 50;
}
```

The (contrived) initialization value for the fourth field, **strength**, is the initialization value of **numLegs** multiplied by 10, and added to 50 (*i.e.*, 90). Note that the initialization expression uses another field defined within the class. This is only legal if the referenced field is defined above the current field, since field initialization expressions are evaluated in the order in which they are defined in the class. Therefore the ordering of fields within the class matters. Also, the initialization expression is only evaluated once when the object is first created and used to assign a starting value to the field. If **numLegs** is changed after the object is created, the initialization expression for **strength** is not re-evaluated.

If a class extends another class, it can also reference any inherited fields:

```
class Tiger extends Animal {
    int tigerStrength = strength * 2;
}
```

Inherited fields are always evaluated before non-inherited fields, with the fields from the higher inherited classes (*i.e.*, closer to the root of the class hierarchy tree) evaluated earlier than fields from lower inherited classes (*i.e.*, farther from the root of the class hierarchy tree). When a **Tiger** object is created, the fields of **Object** are evaluated first, followed by the fields of **Animal**, and finally followed by the fields of **Tiger**.

Inherited fields in an extended class can be redefined. For example, we can do the following:

```
class Tiger extends Animal {
    int tigerStrength = strength * 2;
    Tiger mate;
}
```

The new field, **mate**, is treated as a separate instance variable, which essentially hides the inherited field within class **Tiger**. Notice that **mate** does not have to have the same type as the hidden inherited variable (or actually even a conforming type).

Every class implicitly has two special fields. The first variable, **this**, allows the programmer to refer to the current owner object. **this** can be used to pass the owner object to other methods. It can also be used to access members. The second, **super**, is similar to **this**, however, it only allows access to inherited members, not members defined within the class. In particular, **super** allows programmers to access hidden members. Neither **this** or **super** can be altered (it is illegal to assign to **this** or **super**), they are implicitly set when an object is created. In addition, **super** can not be used as a stand-alone variable. It can only be used to reference hidden inherited members.

Unlike in Java, Bantam fields have no modifiers (*e.g.*, **public**, **static**). All fields are instance variables and are **protected**, and therefore field modifiers are not necessary and are actually illegal. Fields can only be accessed from within the class or a subclass of the class. In Java, **protected** fields can be accessed from classes within the same package. Since Bantam does not include packages, **protected** fields cannot be directly accessed by any class that is not a subclass of the class where the field is defined. Accessing fields from other classes can only be done via methods (which are **public**). There is also no **final** modifier for indicating to the compiler that a field is constant. It is the responsibility of the programmer to guarantee that a constant field is never changed.

The use of protected variables is more restricted in Bantam Java than in Java. In Java, a protected variable can be accessed via an object variable (*e.g.*, a field variable) so long as that variable's type is the same or a super type of the class containing the object reference. In order to simplify the implementation of the compiler, Bantam Java disallows this usage. Therefore, the following is illegal in Bantam Java:

```
class Tiger extends Animal {
    int tigerStrength = strength * 2;
    Tiger mate = ... // initialization not shown
    // LINE BELOW IS ILLEGAL IN BANTAM JAVA
    int mateStrength = mate.tigerStrength;
}
```

In the code above, accessing **tigerStrength** using the object reference **mate** is illegal. In Bantam Java, only **this** and **super** can be used as an object reference in accessing a protected field. As

discussed below, there are no restrictions on performing method calls as methods are public. In the code above, we could create a public method for accessing **tigerStrength** and use this method in the assignment to **mateStrength**.

For reasons having to do with garbage collection, the maximum number of combined inherited and defined fields cannot exceed 1500. This restriction is not a part of the language (and could change in the future, although the maximum would only be made larger not smaller), but rather a part of the particular implementation of the language.

Methods. The format of a method definition is:

```
<type> <name> ( <parameters> ) {  
    <statements>  
    return [ <expression> ] ;  
}
```

where **<type>** is the return type of the method (*e.g.*, **int**), **<name>** is the name of the method, **<parameters>** is a (possibly empty) list of comma-separated formal parameters to the method, and **<statements>** is a (possibly empty) list of statements, which will be executed when the method is called. The last line in the method is always the return statement. This statement starts with the keyword **return** and can be followed by an optional expression. The return statement is terminated with a semi-colon. Like fields, each method defined within a single class must have a unique name (as discussed below, method overloading is not allowed). Method names follow the same conventions as field names.

Below is an example of method **getStrength** in class **Animal**:

```
class Animal {  
    <field definitions (includes strength)>  
  
    int getStrength() {  
        return strength;  
    }  
}
```

The method **getStrength** takes no formal parameters and returns the value of the field **strength** (since the expression in the return statement is **strength**). This method is not contrived since fields are implicitly **protected** and can only be accessed from other classes via a **public** method. The method contains no statements beyond the final return statement. Because **strength** is an **int**, the

return type of **getStrength** is **int**.

Methods can also return nothing by declaring the return type as **void**. In this case, the return statement should not return an expression.

Here is another method in **Animal**, which is **void**:

```
class Animal {
    <member definitions>

    void fight(int amount, boolean isWinner) {
        if (isWinner)
            strength = strength + amount * 5;
        else
            strength = 0;
        return;
    }
}
```

Notice that the method **fight** has two formal parameters: **amount** and **isWinner**. A *formal parameter* is a variable declaration (like a field declaration without the initialization expression). The first formal parameter in **fight**, **amount**, is an **int** variable, indicating the amount to fight. The second formal parameter, **isWinner** is a **boolean** variable indicating whether the animal is the winner of the fight. During a call to the method, each formal parameter is assigned a value called an *actual parameter* passed from the call site. Because each call site can specify **amount** and **isWinner**, **fight** can be used to fight any numeric amount and it can be used both when the animal wins or loses the fight. Parameters effectively allow programmers to write general-purpose methods, which can be called in different contexts. In Section 6 we will look at calling a method.

The method **fight** uses the two parameters to update the **strength** field in the animal object. The statement list in **fight** contains an if statement, which will be discussed in Section 5, but for now it is enough to know that the statement increments the **strength** by 5 times the parameter **amount** if the animal is the winner, otherwise, it sets **strength** to 0. No value is returned (*i.e.*, no expression in the return statement) so the method must be declared **void**.

In Java, the *signature* of a method is the method name combined with the ordered list of parameter types. In the definitions above, the signature of **getStrength** is **getStrength()** and the signature of **fight** is **fight(int,boolean)**. Methods within the same class, in Java, need only have a unique signature and not necessarily a unique name. This feature is called *method overloading*. In Bantam, method overloading is illegal and each method must have a unique name. However, fields and

methods can share names. It is up to the compiler to determine, which is being referenced (the field or method) based on the context in which the name is used.

In the definitions of **getStrength** and **fight**, the type in the return statement matches the declared return type (**int** and **void**, respectively). However, that is only necessary when the declared return type is primitive or **void**. If the declared return type is instead an object type (*i.e.*, a class name), then the type of the return expression only has to conform to the declared return type.

```
class Animal {  
    <member definitions>  
  
    Animal mate;  
  
    Object getMate () {  
        return mate;  
    }  
}
```

The variable **mate** was defined with type **Animal**, however, the method **getMate** is declared to return type **Object**. This method definition is legal since **Animal** is a subclass of **Object** (although in this case we would probably want to specify **Animal** as the return type).

There is one required method in any Bantam Java program: the **main** method within the **Main** class. This method is the starting point of a Bantam Java program. The **main** method must be declared void and define no parameters. Omitting the **main** method or defining **main** in any other way is an error.

Unlike fields, there are no restrictions on the number of inherited and defined methods within a particular class. Like fields, methods do not have modifiers. All methods are **public**, meaning they can be called from within any class (given a dispatch object as discussed in 6). Like fields, methods can be redefined, however, a redefined method must have the same signature and return type as the inherited method.

Let's now take a closer look at the statements that we can use within a method.

5 Statements

There are five types of statements in Bantam Java. We discuss each below.

Block statement. The first statement, a block statement, allows us to insert multiple statements

where exactly one is needed. The format of a block statement is:

```
{ <statements> }
```

The block statement begins and ends with ‘{’ and ‘}’, respectively. Inside the braces are a list of statements. These statements can include potentially another block statement, *i.e.*, a block statement nested within another block statement. The utility of the block statement will become clear when we look at the other types of statements.

Declaration statement. A declaration statement declares a new variable and assigns it an initial value. This new variable is called a *local* variable since it may be accessed only within the method (there are other rules for accessing the variable, which are discussed below). The format of a declaration statement is:

```
<type> <name> = <expression> ;
```

where **<type>** and **<name>** are the type and name of the declared variable, respectively, and **<expression>** is the initialization expression of the variable. Declaration statements must be terminated with a semi-colon. The rules and conventions for declaration statements are similar to fields except that the initialization expression is not optional. The initialization expression is not required in Java, however, Java requires that all local variables are assigned before they are used. Determining whether the variable is assigned before its first use requires some advanced compiler techniques, which we wanted to avoid in the Bantam compiler. We get around this by requiring all locally declared variables to be initially assigned.

Below is an example of a declaration statement:

```
class Main {  
    void main () {  
        int x = 3;  
        ...  
        return;  
    }  
}
```

In this case, we have declared a new variable **x**, which is local to the method **main**. The variable **x** cannot be used outside of the method **main**. In fact, it can only be used from the point it was defined

until the end of the innermost enclosing block statement or the end of the method, whichever comes first. This range is called the *scope* of x . In this way, methods and block statements define new scoping levels. As in Java, a local variable can be defined using the same name as a field defined in the same class or an inherited class. While in the local variable's scope, the field is hidden, although it can still be accessed via **this**. Two local variables defined within the same method can also use the same name as long as they do not have overlapping scopes. It is a compiler error to declare two local variables with overlapping scopes using the same name.

The code below shows an example in Bantam Java of some variable declarations. Note: the subscripts are not a part of the program, but instead, used to help the reader differentiate between block statements and variables with the same name.

```
class Main {
  int x0 = 0;
  void main () {
    int a = x0;
    {b1
      int b = x0;
      int x1 = 1;
      int c = x1;
      {b2
        // this would be illegal:
        // int x2 = 2;
        int d = x1;
      }b2
    }b1
    {b3
      int e = x0;
      int x3 = 3;
      int f = x2;
    }3
  }
  return;
}
```

The variable x_0 is a field and its scope is the entire class **Main**. The assignment to **a** will use x_0 (it will be assigned to the value 0) since no local variable x has yet been declared within **main**. The variable x_1 is a local variable and its scope is from its declaration until the end of the first block statement (block statement labeled with subscript **b1**) within **main**. The assignment of **b** comes before the declaration of x_1 so **b** is assigned the value of x_0 (*i.e.*, 0). The assignment of **c** comes after the declaration and within the first block statement so **c** is assigned the value of x_1 (*i.e.*, 1). Within the first block statement, there is a second block statement (labeled with subscript

b2). Within this second block statement, it would be illegal to declare another x (x_2) since the two variables would have overlapping scopes. The variable d is assigned the value in x_1 since it is still within the first block statement and it appears after the declaration of x_1 . The variable x_3 is another local variable like x_1 but it has a disjoint scope. Its scope is from the declaration until the end of the third block statement (labeled with subscript **b3**). The assignment of e comes before the declaration of x_3 so e is assigned the value of x_0 (*i.e.*, 0). The assignment of f comes after the declaration of x_3 and within the third block statement so f is assigned the value in x_3 (*i.e.*, 3). Notice that x_0 can still be accessed anywhere in the method via **this**.

If statement. As in Java, an if statement allows the programmer to branch between statements based on some predicate. The format of an if statement is:

```
if ( <predicate> )
  <then statement>
[ else
  <else statement> ]
```

where **if** and **else** are keywords, the **<predicate>** is a **boolean** expression, the **<then statement>** is a statement that is evaluated if the predicate evaluates to **true**, and the **<else statement>** is a statement that is evaluated if the predicate evaluates to **false**. As in Java, the else statement is optional. Here is an example if statement:

```
class Main {
  void main () {
    ...
    if (animal.getStrength() > 100)
      animal.fight(100, true);
    else
      animal.fight(100, false);
    ...
    return;
  }
}
```

Only one statement can be used as the then or else statement. However, if we need to execute more than one statement in either of these (or both) we can use a block statement:

```

class Main {
    void main () {
        ...
        if (animal.getStrength() > 100) {
            animal.fight(100, true);
            strength = animal.getStrength();
        }
        ...
        return;
    }
}

```

If more than one predicate needs to be tested, if statements can be cascaded as they are in Java. For example:

```

class Main {
    void main () {
        ...
        if (animal.getStrength() > 100)
            animal.fight(100, true);
        else if (animal.getStrength() > 50)
            animal.fight(50, false);
        else if (animal.getStrength() > 10)
            animal.fight(10, false);
        else
            animal.fight(0, false);
        ...
        return;
    }
}

```

This code is actually made up of three if statements not one. The last two if statements are nested within the previous if statement's else clause.

Unlike Java, Bantam does not have switch statements, however, these can often be implemented using cascaded if statements like those in the code above.

As shown in the example below, a potential ambiguity can arise due to the optional else statement:

```

if (x > y)
    z = x;
if (r > s)
    t = r;
else
    t = s;

```

In the code above, the else could potentially bind to the first or second if statement. As in Java, an else always binds to the innermost, unclosed if. Therefore, the else in the code above binds to the second if (*i.e.*, it is executed if and only if $r \leq s$).

While statement. A while statement can be used to repeat a statement some number of times. The format of a while statement is:

```
while ( <predicate> )
    <statement>
```

where **<predicate>** is a boolean expression, and **<statement>** is executed while the predicate evaluates to **true**. The while loop is evaluated by first evaluating the predicate, and if this is **true** then evaluating the statement, and repeating the process. The loop stops once the predicate evaluates to **false**. Each time the statement within the loop executes is called a loop *iteration*.

Here is an example, while loop:

```
class Main {
    void main () {
        ...
        while (animal.getStrength() < 100)
            animal.fight(100, true);
        ...
        return;
    }
}
```

This code repeatedly checks whether the **animal.getStrength()** is less than 100. If it is less than 100, then **animal.fight(100, true)** is executed. If it not less than 100, then the loop terminates.

Like if statements, the statement within a while loop can be replaced with a block statement.

Bantam does not include any other types of loops such as for loops or do...while loops, however, these are easily converted into while loops. Below is a Java for loop and the corresponding Bantam while loop:

```
// Java for loop
for (int i = 0; i < 100; i = i + 1)
    animal.fight(10, true);
```

```
// corresponding loop in Bantam
int i = 0;
while (i < 100) {
    animal.fight(10, true);
    i = i + 1;
}
```

Expression statement. An expression statement is a statement made up of a single expression (which itself may contain subexpressions) and terminated by a semi-colon. Although we save the discussion of expressions until the next section, here is an example expression statement using a simple assignment expression (the variable on the left side of ‘=’ is set to the computed value on the right side):

```
x = y + 2;
```

Not all expressions can be used within an expression statement. In fact, the only expressions that are legal within an expression statement are assignments, method calls, and new object constructions. The code below is not a legal expression statement:

```
foo() + 2;
```

The addition of 2 to **foo()** is a useless operation in this case since it is not assigned to any variable, and for this reason, this is an illegal expression statement.

Let’s now look at the last category of language constructs in Bantam: expressions.

6 Expressions

Bantam supports a variety of expressions, which are described below. Figure 1 lists the operators that are supported in Bantam. Bantam supports most Java operators including arithmetic operators (+, -, *, /, and %), relational operators (==, !=, <, <=, >, and >=), boolean operators (!, &&, | |), and assignment (=). Bantam also supports object construction, member reference, (object) casting, and **instanceof**. We discuss each of these in more depth below.

Operator types	Operators
Arithmetic	+ - * / %
Relational	== != < > <= >=
Boolean	! &&
Assignment	=
Cast	(type)(expression)
Instanceof	instanceof
Member reference	.
Object construction	new

Table 1: Bantam Java operators.

Every expression in Bantam computes a value with some data type. The type of the computed value is the type of the expression. Because expressions are used within field declarations, statements, or other expressions, the type of the expression is often important. For example, an expression that computes a value of type **int** cannot be assigned to a **boolean** field. It is the responsibility of the compiler to determine the type of each expression and verify that it is used in the appropriate way. For each expression below, we will discuss both what the expression computes and its resulting data type.

Assignment. An assignment expression has the form:

$$[\text{<reference> . }] \text{<name> = <expression>}$$

where **<reference>** is an optional reference object (followed by ‘.’), **<name>** is the name of the target variable, and **<expression>** determines the value to assign to the variable. The target must refer to some variable defined in the same scope or an encompassing scope. The reference object can be used only in the assignment of a field. Because fields are **protected** (and **protected** is more restrictive in Bantam Java than Java), the reference object, if used, must be either **this** or **super**.

Here is an example assignment:

$$x = y + 2;$$

The expression on the right is evaluated first and then stored in the lefthand variable. The result of the entire expression is the result of the righthand expression.

If the type of the righthand expression or the lefthand variable is primitive then the types must match, otherwise the (object) type on the right must conform to the (object) type of the lefthand

variable. The static type of the entire expression is the static type of the righthand expression.

Dynamic dispatch. A dynamic dispatch is simply a method call of an instance method. To call a method in Bantam, we need a **reference object** (of an appropriate type) to invoke the method on. The reason for the name *dynamic dispatch* is that in object-oriented languages a method call of an instance method is often thought of as passing a message to the object, hence the word “dispatch” (the “dynamic” part will be discussed shortly). Notice that because the called method can access fields, the outcome of the method depends on the reference object.

The format of a dynamic dispatch is:

$$[\text{<reference> . }] \text{<name> (<parameters>)}$$

where **<reference>** is an optional expression that corresponds to the reference object, **<name>** is the name of the method, and **<parameters>** is a list of comma-separated actual parameters, each of which is an expression. If the reference expression is not included, then **this** is used as the reference object.

In order for the dynamic dispatch to be legal, the called method must be declared within the reference object’s static type class or within a superclass. The number of actual parameters and declared formal parameters must match and the type of each actual parameter expression must conform to the type of the declared formal parameter or match if either type is a primitive. Finally, the dynamic dispatch must be used according to its declared return type. For example, a **void** method should not be used within a larger expression since it has no return value. A method that returns a **boolean** cannot be used in an arithmetic expression where an **int** is required.

The following code below shows how to dispatch on methods **fight** and **getStrength** within class **Animal** (defined on page 10):

```
class Main {
    Animal animal = new Animal();

    void main() {
        animal.fight(20*5, true);
        int halfStrength = animal.getStrength() / 2;
        ...
        return;
    }
}
```

The dispatch on **fight** passes an **int** (**20*5**) as the first parameter and a **boolean** (**true**) as the second parameter, which match the declared formal parameters of **fight**. **fight** is a **void** method so it must appear on a line by itself and not within a larger expression. The method **getStrength** takes no parameters and returns an **int**. Therefore, we can use the result of **getStrength** in an expression that requires an **int**.

The actual parameter expressions in a dynamic dispatch are executed from left to right and passed to the method. They are evaluated after the reference expression. Bantam Java, like Java, uses *call by value* meaning that the formal parameter in the method is assigned the value from the corresponding actual parameter expression. In the code above, the formal parameter **amount** is set to **20*5** (*i.e.*, **100**) and the formal parameter **isWinner** is set to **true**.

Below is another example of a (contrived) method call. In the call to **foo** from method **main**, because of call by value, the formal parameter **y** is set to the value in **main**'s variable **x**. But **x** and **y** are separate variables. If **foo** sets **y** to a new value, this will have no effect on **x** in **main**.

```
class Main {
    void main() {
        int x = 3;
        foo(x);
        return;
    }

    void foo(int y) {
        ...
    }
}
```

The parameter in the code above has a primitive type (*i.e.*, **int**). Call by value is also used for parameters with object types. However, as variables in Bantam Java store references (or memory pointers) to objects rather than the objects themselves, it is the reference that is copied not the object itself. For example, in the code below, the **animal** reference is passed to method **foo**.

```

class Main {
    void main() {
        Animal animal = new Animal();
        foo(animal);
        return;
    }

    void foo(Animal a) {
        ...
    }
}

```

While in **foo**, the variables **a** and **animal** (which is not actually accessible in **foo**) refer to the same object (*i.e.*, alias). If we modify a field of **a**, then the field will be changed for **animal**. Of course, if we assign **a** to another **Animal** object, then the two variables would no longer alias.

Because of inheritance, the exact method that is called on a dispatch is not always known at compile time, which is the reason for the word “dynamic” in dynamic dispatch. This property of object-oriented languages, *i.e.*, that one call site can call multiple methods during the course of a program, is known as *polymorphism*. For example, the class **Tiger** could define its own version of **fight**:

```

class Tiger {
    <member definitions>

    void fight(int amount, boolean isWinner) {
        if (isWinner)
            strength = strength + amount * 10;
        else
            strength = 0;
        return;
    }
}

```

If the method **fight** is called on an **Animal** object, which **fight** is called? It depends on the runtime type of the reference object, which as discussed above may differ from the static type (*i.e.*, **Animal**). If at runtime, the object has type **Animal**, then the **fight** method within the **Animal** class is called. If, on the other hand, the object has dynamic type **Tiger**, then the **fight** method within the **Tiger** class is called. This example is illustrated in the **Main** class shown on page 19. Although we know the original dynamic type of **animal** is **Animal**, the dynamic type of this field may change later, and in general, it is impossible for the compiler to determine the dynamic type of a variable. The compiler must generate code for calling the appropriate method at runtime based

on the dynamic type of the reference object. A runtime error is generated if the reference object is **null**.

The reference expression can evaluate to any object type (so long as that type includes the appropriate method) or can refer to the special variables **this** or **super**. Using **this** is not necessary since excluding the reference object has the same effect. However, using **super** is often helpful. It allows the programmer to call hidden methods. For example, in the **Tiger** class, if we want to call the **fight** method from **Animal**, we can only do this using **super** as the reference object.

Object construction. We can use the **new** operator to create a new object from a class (called *object construction*). The format of a **new** operation is:

```
new <class name>()
```

where **new** is a keyword and **<class name>** is replaced with a class name. For example, if we wanted to create an object from a class **Tiger** and store it in a variable of type **Tiger** we could do the following:

```
Tiger t = new Tiger()
```

In Java, the programmer can define their own constructors that are called when the object is created. These allow the programmer to customize the object based on the particular context in which it is created. User-defined constructors are not allowed in Bantam. Object construction in Bantam creates a new object and initializes each field based on the initialization expression (or lack of). However, a Bantam programmer can write methods that simulate user-defined Java constructors. Of course, it is up to the programmer to make sure that the pseudo-constructor is called after every object construction. For example, in the **Animal** class, we could define a pseudo-constructor to set the number of legs and the flag indicating whether the animal can fly.

```

class Animal {
    <member definitions>

    // simulates a constructor
    // must always be called immediately after construction
    Animal init(int l, boolean f) {
        numLegs = l;
        canFly = f;
        return this;
    }
}

class Main {
    void main() {
        Animal animal = (new Animal()).init(4, false);
        ...
    }
}

```

Casting. As we have already seen, we can store an object of one type into a variable whose type is a super type. For example, we can store a **Tiger** object into a variable with type **Animal**. This action is called *upcasting*, since we are moving up the class hierarchy tree. Upcasting can be done either implicitly or explicitly. For example, the following two statements are equivalent:

```

Animal animal = new Tiger();
OR
Animal animal = (Animal)(new Tiger());

```

In the first assignment statement, we implicitly upcast a **Tiger** object into an **Animal** so that it can be stored in **animal**. In the second assignment statement, we do the same except we explicitly show the upcast. The format of an explicit cast is as follows:

```
( <class name> ) ( <expression> )
```

where **<class name>** is the name of some existing class and **<expression>** is an expression with an object type (*i.e.*, not a primitive). In Bantam, unlike in Java, the expression to be casted must be enclosed by parentheses (this simplifies parsing).

In the example cast above (*i.e.*, **(Animal)(new Tiger())**) only the static type of **animal** is **Animal**. The runtime type is still **Tiger**. The additional **Tiger** functionality (non-inherited members) is still maintained in the variable **animal**, however, it cannot be utilized. But it can be recovered by

performing a *downcast*. For example, we could do the following:

```
Tiger tiger = (Tiger)(animal);
```

In this case, the cast must be explicit since it needs to be checked at runtime, *i.e.*, the compiler does not know statically whether **animal** is actually a **Tiger** or a subclass of **Tiger**. If the dynamic type of **animal** is not **Tiger** or a subclass of **Tiger** (*e.g.*, **Bear**) then a runtime error occurs.

Sometimes the compiler can statically catch an illegal cast, *i.e.*, a cast that is neither an upcast or a downcast. In particular, any cast from one class to another class, which is neither a subclass or a superclass, is a compiler error. For example, the following is a compiler error:

```
Bear bear = (Bear)(new Tiger());
```

Upcasting and downcasting are common operations in Bantam (as well as in Java, prior to the addition of generics). Upcasting and downcasting make it possible to create collections of heterogeneous objects like the **Vector** class in Java. The collection is simply an object that stores each element as an **Object**, *i.e.*, objects are upcasted to **Object**. Because **Object** is a superclass of all classes, the collection can house any kind of object (but not a primitive). When an object is retrieved from the collection, it usually must be downcasted from **Object** to its original type. (Note: because the Bantam Java language does not have arrays, collections in Bantam would need to rely on linked lists to store elements, which may not be all that efficient.)

Bantam Java, unlike Java, does not have support for casting primitives. Therefore, it is a compiler error if the programmer attempts to cast to a primitive type or cast an expression, which has a primitive type. A compiler error also occurs if the target type does not exist.

Instanceof. The **instanceof** operation in Bantam allows the programmer to determine the runtime type of an object. Like casting, **instanceof** is often a useful operation when dealing with collections of heterogeneous objects. If we pull an object off of a collection, such as a **Vector**, we may not know its exact type, but rather that the object is one of several types. The format of **instanceof** is:

```
<expression> instanceof <class name>
```

where **<expression>** is an expression with an object type (*i.e.*, not a primitive) and **<class name>**

is the name of an existing class. **instanceof** evaluates to **true** if the dynamic type of the expression conforms to the righthand type. Otherwise, it evaluates to **false**.

The following code illustrates a common use of **instanceof**. The code retrieves a generic **Animal** object from a list (via the method **getNextAnimal**). The **instanceof** operator is used to determine the specific type of **Animal** (**Tiger**, **Bear**, or **Eagle**) so that a counter can be updated, which keeps track of the total number of each kind of **Animal**. (Note: if the dynamic type of **animal** is **Animal** then none of the **instanceof** expressions will evaluate to **true** and none of the counters will be incremented.)

```
Animal animal = getNextAnimal();
if (animal instanceof Tiger)
    numTigers = numTigers + 1;
else if (animal instanceof Bear)
    numBear = numBear + 1;
else if (animal instanceof Eagle)
    numEagles = numEagles + 1;
```

A compiler error occurs if the lefthand expression in the **instanceof** expression has a primitive type. An error also occurs if the righthand type in the **instanceof** is primitive or does not exist. As with casting, a compiler error also occurs if the expression's type is neither a subtype nor a supertype of the righthand type.

Arithmetic operators. Bantam supports five binary arithmetic operators (+, -, *, /, %) and one unary arithmetic operator (-). The format of the binary operators is:

<left expression> <operator> <right expression>

where <left expression> and <right expression> are both expressions with type **int** and <operator> is one of '+', '-', '*', '/', or '%'. The operators are similar to those in Java. + computes the sum of the left and right expressions, - computes the difference, * computes the product, / computes the integer division, and % computes the remainder when dividing the left expression by the right expression. The resulting type of these expressions is an **int**.

The format of the unary operator is:

- <expression>

where **<expression>** is an **int** expression. This operation computes the arithmetic negation of the expression. The resulting type is an **int**.

Here are some examples:

```
int x = 0;
int y = 1;
int z = 2;

x = y + z; // x is set to 3
x = y - z; // x is set to -1
x = y * z; // x is set to 2
x = y / z; // x is set to 0
x = y % z; // x is set to 1
x = - y; // x is set to -1
```

Boolean operators. Bantam supports two binary boolean operators (**&&** and **||**) and one unary boolean operator (**!**). The format of the binary operators is:

<left expression> <operator> <right expression>

where **<left expression>** and **<right expression>** are both expressions with type **boolean** and **<operator>** is one of '**&&**' or '**||**'. **&&** computes the logical AND of the left and right expressions. If both the left and right expressions evaluate to **true**, then the result of **&&** will also be **true**. Otherwise, the result is **false**. **||** computes the logical OR of the left and right expressions. If both the left and right expressions evaluate to **false**, then the result of **||** will also be **false**. Otherwise, the result is **true**. The resulting type of these expressions is a **boolean**.

The format of the unary operator is:

! <expression>

where **<expression>** is a **boolean** expression. **!** computes the complement of the expression. If the expression evaluates to **false**, the result is **true**. If the expression evaluates to **true**, the result is **false**. The resulting type of these expressions is a **boolean**.

Here are some examples:

```
boolean b1 = false;
boolean b2 = false;
boolean b3 = true;

b1 = b2 && b3; // b1 is set to false
b1 = b2 || b3; // b1 is set to true
b1 = !b2; // b1 is set to true
```

Relational operators. Bantam supports six binary relational operators (==, !=, <, <=, >, >=) for comparing two values. The format of the relational operators is:

<left expression> <operator> <right expression>

If <operator> is '==' or '!=' then the two expressions must have the same type if either is primitive or one type must conform to the other type if either is an object. If <operator> is '<', '<=', '>', or '>=' then the expressions must both have type **int**. The result of a relational operation is a **boolean** indicating whether the test succeeded.

The == operator tests if the left and right expressions are equivalent. Note: if the expressions are objects the operator tests if the left and right expression refer to the exact same object. If they refer to different objects then the test fails, even if the objects have the exact same field values. If the left and right expressions are equivalent then the result of == is **true**, otherwise it is **false**. The != operator works like == except that it tests if the left and right expressions are not equivalent. If the left and right expressions are equivalent then the result of != is **false**, otherwise it is **true**.

The < operator tests if the lefthand **int** is less than the righthand **int**, <= tests if the lefthand **int** is less than or equal to the righthand **int**, > tests if the lefthand **int** is greater than the righthand **int**, and >= tests if the lefthand **int** is greater than or equal to than the righthand **int**. If the test succeeds then the result is **true**, otherwise the result is **false**.

Here are some examples:

```

boolean b = false;
int i1 = 0;
int i2 = 1;
int i3 = 0;
Object o1 = new Object();
Object o2 = new Object();
Object o3 = o1;

b = i1 == i2; // b is set to false
b = i1 != i2; // b is set to true
b = o1 == o2; // b is set to false
b = o1 == o3; // b is set to true
b = i1 < i2; // b is set to true
b = i1 >= i3; // b is set to true

```

Constants. There are three constant types in Bantam: **int**, **boolean**, and **String**. An **int** constant is any sequence of numeric digits (*e.g.*, **0**, **124**). It must be between **0** and **2147483647** ($2^{31} - 1$) since it is stored as a 32-bit signed integer. Integer constants must be written in decimal, other forms such as hexadecimal are not supported. Leading 0's are allowed in an **int** constant.

A **boolean** constant is either **true** or **false**. These are keywords and may not be used as identifiers.

A **String** constant is a sequence of characters between double quotes (*e.g.*, "abc"). Unlike in Java, Bantam Java represents characters using an ASCII encoding rather than an Unicode encoding, which limits the types of characters that can be represented in a Bantam Java **String** constant. As in Java, **String** constants are treated as objects of type **String** (which is discussed in 7). **String** constants may not span multiple lines. A backslash is interpreted as an escape character within a **String** constant, which allows the programmer to specify some special characters. These include `\n` (newline), `\t` (tab), `\"` (double quote), `\\` (backslash), and `\f` (form feed). Other special characters are illegal. In particular, a backslash within a **String** constant must be followed by 'n', 't', 'f', '\', or a double quote. For garbage collection reasons, the size of a **String** constant (and a **String** variable) is limited to 5000 characters.

Variables. A variable reference is also an expression. Of course, the variable must have been previously declared, otherwise, it is an error. If the variable is a field, then it can be preceded by **this** followed by `.'.` If the variable is an inherited field (hidden or otherwise) then it can be preceded with **super** followed by `.'.`

Parentheses. The final type of expression is an expression within parentheses: (<**expression**>

., (params)
- (unary), !
new, cast
*, /, %
+, -
<, >, <=, >=, instanceof
==, !=
&&
=

Table 2: Operator precedence and associativity. Operators with higher precedence are listed higher in the table than those with lower precedence. Operators in the same row have the same precedence. All binary operators are left-associative except assignment ('='). These rules were taken from the Java book by Arnold, Gosling, and Holmes (Gosling led the team at Sun, which created Java) [1]. Note: (params) refers to the parentheses and actual parameter list in a method call.

). Parentheses are useful for grouping operations. Table 2 shows the order of operations and the associativity for each operation discussed in this section. Parentheses should be used to override precedence and/or associativity. For example, we could do the following to force the addition to occur before multiplication:

```
int x = (2 + 3) * 5; // x is set to 25
```

7 Built-In Classes

There are four built-in classes in Bantam. These are listed in Table 3.

Object. The **Object** class is similar to the **Object** class in Java except with less functionality. All classes are a subclass of **Object**. **Object** contains one method called **clone**, which copies an object. If the cloned object contains fields (inherited or otherwise) then the values of these fields are copied to the new object. If these fields are themselves objects, however, the contents of those objects are not copied. In this case, the field in both the original and the copy would refer to the same object. To overcome this limitation, one can always redefine the **clone** method when defining a new class. The **Object** class is the only built-in class that can be extended, and in fact, all classes must either

Class	Method signature	Method description
Object	Object clone()	copy an object
Sys	void exit(int status)	exit program with specified status
	int time()	return UTC time
	int random()	return random integer
String	int length()	return string length
	boolean equals(Object s)	test if strings are equivalent
	String substring(int beginIndex, int endIndex)	return substring between the indices
	String concat(String s)	return concatenated string
TextIO	void readfile(String filename)	set to read from specified file
	void writefile(String filename)	set to write to specified file
	void readStdin()	set to read from standard input
	void writeStdout()	set to write to standard output
	void writeStderr()	set to write to standard error
	String getString()	read next string
	int getInt()	read next int
	void putString(String s)	write specified string
	void putInt(int i)	writes specified int

Table 3: Bantam built-in classes. Note: **Sys**, **String**, and **TextIO** cannot be extended.

directly or indirectly extend **Object**.

Sys. The **Sys** class contains three methods for exiting the program, getting the UTC time in seconds, and computing a random integer. It extends **Object**. The method **exit** takes one parameter, an **int** representing the exit status (0 is generally used for no error, non-zero values are used when an error has occurred). The method **time** takes no parameters and returns an **int** value representing the seconds that have passed since 1970 using Greenwich Mean Time (GMT). The method **random** takes no parameters and returns a random 32-bit **int**. Note: the **Sys** class cannot be extended.

String. The **String** class is a class for representing sequences of characters. It extends **Object** and cannot be extended. It is similar to the **String** class from Java, but with less functionality. For reasons having to do with garbage collection, the length of a **String** may not exceed 5000 characters. The characters in a **String** are represented using ASCII rather than Unicode, which limits the types of characters that can be stored in a Bantam Java **String**.

String has four methods: **length**, **equals**, **substring**, and **concat**. The method **length** takes no parameters and returns the length (an **int**) of the **String**. For example, if the variable **s** is a **String** representing “abc”, then calling **s.length()** returns 3.

The method **equals** is a method for comparing two strings. It takes one parameter: an **Object** to

compare to the reference **String**. If the parameter has the runtime type **String** and it represents the same sequence of characters as the reference string then **equals** returns **true**. Otherwise, it returns **false**. A runtime error occurs if the **String** parameter is **null**. Notice that this method behaves differently than **==**. The operator **==** can only determine if the two strings refer to identical objects. It cannot be used to determine if two strings represent the same sequence of characters.

The method **substring** takes two parameters: a beginning index (an **int**) and an end index (an **int**). It returns the **String** that starts at the beginning index and ends at the end index. The character at the beginning index is included in the resulting **String**, but the character at the end index is not included in the **String**. If **s** is a **String** representing “abc”, then calling **s.substring(1,3)** returns a **String** representing “bc”. A runtime error occurs if the beginning index is less than 0, the end index is greater than the length of the **String**, or the beginning index is greater than the end index.

Finally, the method **concat** takes as parameter a **String** and returns a new string representing the concatenation of the reference **String** object with the parameter **String** object. For example, if **s** is a **String** representing “abc” and **s2** is a **String** representing “def”, then **s.concat(s2)** would return a new **String** representing “abcdef”. A runtime error occurs if the **String** parameter is **null** or if the resulting **String** has more than 5000 characters.

There is no method for copying strings, but because **String** extends **Object**, the **clone** method can be used to copy a **String**.

TextIO. The **TextIO** class provides methods for performing file input and output. It extends **Object**. It is similar to the **TextIO** class defined by David Eck in his textbook [2], except the class does not use **static** members. Reading and writing are separately handled by this class and so it is possible with one **TextIO** object to read from one file, while writing to another. The **TextIO** class contains a method called **readFile** for setting the read file. This method takes as input a **String** representing the file name. It terminates the program if the file does not exist or the program does not have permission to read from the specified file. There is also a method called **writeFile** for setting the write file. This works similarly to **readFile** except the specified file need not exist. If the file does not exist, then **writeFile** will create the specified file. The program does, however, have to have permission to write to the specified file. For both **readFile** and **writeFile** the programmer can specify a path along with the file name, *e.g.*, “./file.txt” or “/home/someuser/foo”. A runtime error occurs in either method if the input **String** is **null**.

TextIO also contains methods for reading from standard input (**readStdin**) and writing to ei-

ther standard output (**writeStdin**) or standard error (**writeStderr**). These methods do not take any parameters. By default, **TextIO** reads from standard input and writes to standard output.

None of the **TextIO** methods described above actually read or write any data, instead they set either the read location or the write location. To read or write from some location, the programmer can use **getString**, **getInt**, **putString**, or **putInt**. The method **getString** reads the next line of text (minus the newline character) from the current read location and returns a **String** representing this text. The method **getInt** reads the next line of text (minus the newline character) from the current read location and interprets the text as an **int**. If an **int** was entered then **getInt** returns the entered value otherwise it returns 0. The method **putString** takes a **String** parameter and writes it to the current write location. A runtime error occurs in **putString** if the input **String** is **null**. The method **putInt** takes an **int** parameter and writes it to the current write location. Neither **putString** or **putInt** write a newline character to the output stream, however, this can be achieved by performing **putString("\n")** after writing the **String** or **int**.

Example. Figure 2 shows a somewhat contrived example class that uses all of the built-in classes. There is one **Main** class with three methods: **main**, **getNextLine**, and **error**. The **main** method prompts the user via standard input for a positive integer **n** and then calls **getNextLine n** times to read **n** lines from the file “input.txt”. It stops early if the program reads the string “quit”. The method **getNextLine** reads the next line from “input.txt” and calls **error** if either there is not another line to read or if the length of the read string is less than 2. The method **error** prints an error message to standard error and exits the program with status 1. After **main** gets the next line from **getNextLine** it concatenates it with the previous read output. At the end of the method, **main** prints the concatenated output to standard output and to the file “output.txt” along with the current time (represented as the seconds passed since 1970 using GMT) and a random integer.

8 Bantam vs. Java

Although we have described some of the differences between Bantam and Java throughout this manual, we lay out the main differences below. We also show how to convert a Bantam program to a Java program (note: converting a Java program to a Bantam program is a more difficult process, especially if the Java program uses a significant number of features that were excluded from Bantam Java).

```

class Main {
    TextIO io = new TextIO();
    Sys sys = new Sys();
    String output = "";

    void error() {
        io.writeStderr();
        io.putString("Bad input; exiting\n");
        sys.exit(1);
        return;
    }

    String getNextLine() {
        String s = io.getString();
        if (s == null || s.length() < 2)
            error();
        return s.substring(1, s.length());
    }

    void main() {
        String s = "";
        io.readStdin();
        int n = io.getInt();
        if (n < 1)
            error();

        io.readFile("input.txt");
        int i = 0;
        while (i < n && !s.equals("quit")) {
            s = getNextLine();
            output = output.concat(s).concat("\n");
            i = i + 1;
        }

        io.writeStdout();
        io.putString(output);
        io.putString("GMT: ").putInt(sys.time()).putString("\n");
        io.putString("Random: ").putInt(sys.random()).putString("\n");

        io.writeFile("output.txt");
        io.putString(output);
        io.putString("GMT: ").putInt(sys.time()).putString("\n");
        io.putString("Random: ").putInt(sys.random()).putString("\n");

        return;
    }
}

```

Figure 2: Example program using all of the built-in classes.

Differences. The Bantam Java language excludes the following features (this list may not be complete):

- Primitives besides **int** and **boolean**
- Arrays
- Static members
- Packages
- Modifiers such as **public**, **private**, **protected**, and **package**
- Nested classes
- Abstract classes
- Interfaces
- Final variables or methods
- Enumerations
- Generics
- Loops besides while loop
- Break and continue statements
- Switch statements
- Logic operators (&, |, ^, <<, >>, >>>)
- Shortcut operations such as **++**, ***=**, *etc.*
- Conditional expressions
- Hexadecimal/octal representation of **int** constants
- User-defined constructors
- Method overloading
- Arbitrary return statement placement within a method
- Declaration statements without assignments
- Unicode characters in **String** objects (ASCII is used instead)
- Extensive library for graphics, networking, mathematical functions, *etc.*

Converting Bantam programs to Java programs. Because the Bantam language is a subset of the Java language, it is always straightforward to convert Bantam programs into Java programs. The programmer must perform the following steps:

- Add the **protected** keyword in front of all fields
- Add the **public** keyword in front of all methods
- Create classes **TextIO** and **Sys** in Java

- Add the following static **main** method to the **Main** class (without replacing the original **main** method):

```
public static void main(String[] args) {  
    (new Main()).main();  
}
```

Otherwise, the program will not need to be changed.

References

- [1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.
- [2] David Eck. *Introduction to Programming using Java - Part I*. Creative Commons, 5th edition, 2006.